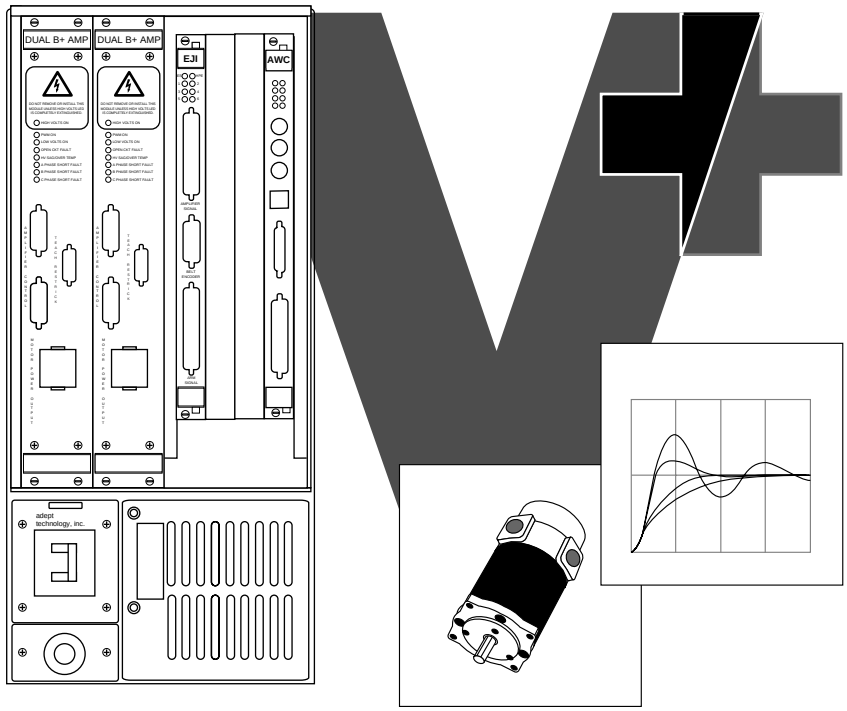
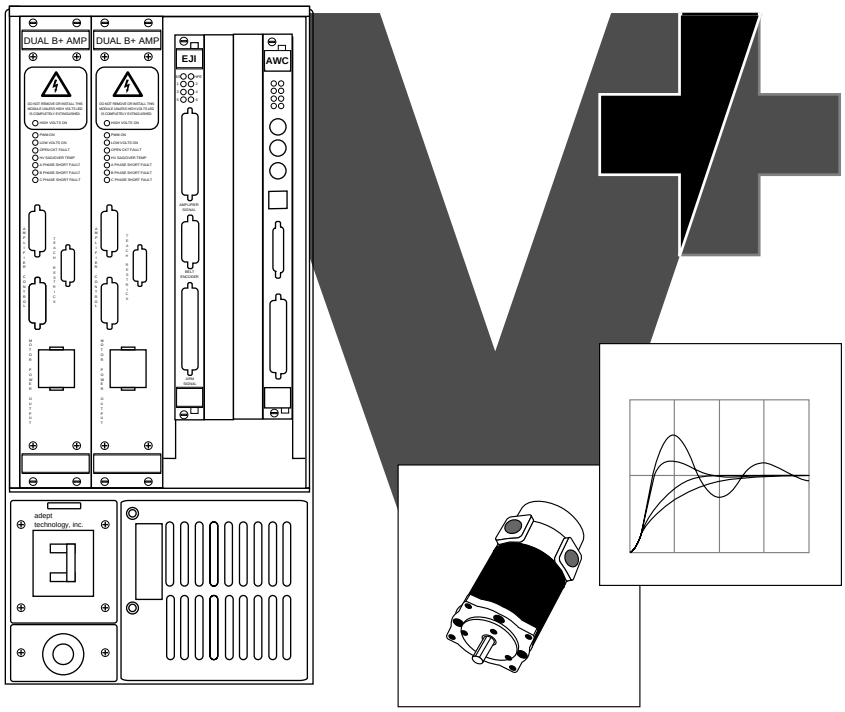
Advanced Servo Library
Reference Guide
Version 13.0



Advanced Servo Library
Reference Guide
Version 13.0



Part # 00963-04000, Rev A
August 1998

adept
technology, inc.

150 Rose Orchard Way • San Jose, CA 95134 • USA • Phone (408) 432-0888 • Fax (408) 432-8707

Otto-Hahn-Strasse 23 • 44227 Dortmund • Germany • Phone (49) 231.75.89.40 • Fax(49) 231.75.89.450

41, rue du Saule Trapu • 91300 • Massy • France • Phone (33) 1.69.19.16.16 • Fax (33) 1.69.32.04.62

The information contained herein is the property of Adept Technology, Inc., and shall not be reproduced in whole or in part without prior written approval of Adept Technology, Inc. The information herein is subject to change without notice and should not be construed as a commitment by Adept Technology, Inc. This manual is periodically reviewed and revised.

Adept Technology, Inc., assumes no responsibility for any errors or omissions in this document. Critical evaluation of this manual by the user is welcomed. Your comments assist us in preparation of future documentation. A form is provided at the back of the book for submitting your comments.

Copyright © 1998 by Adept Technology, Inc. All rights reserved.

The Adept logo is a registered trademark of Adept Technology, Inc.

Adept, AdeptOne, AdeptOne-MV, AdeptThree, AdeptThree-XL, AdeptThree-MV, PackOne, PackOne-MV, HyperDrive, Adept 550, Adept 550 CleanRoom, Adept 1850, Adept 1850XP, A-Series, S-Series, Adept MC, Adept CC, Adept IC, Adept OC, Adept MV, AdeptVision, AIM, VisionWare, AdeptMotion, MotionWare, PalletWare, FlexFeedWare, AdeptNet, AdeptFTP, AdeptNFS, AdeptTCP/IP, AdeptForce, AdeptModules, AdeptWindows, AdeptWindows PC, AdeptWindows DDE, AdeptWindows Offline Editor, and V+ are trademarks of Adept Technology, Inc.

Any trademarks from other companies used in this publication are the property of those respective companies.

Printed in the United States of America

Table of Contents

1 Overview. 9

Introduction 10
Compatibility 11
Adept Robot Access 11
Related Publications 12
Installation 12
Summary of Library Routines 12
Initialization and Other Utility Routines 12
Robot Motion Routines 13
Status Routines 13
Routines for Collecting Servo Data 13
Encoder and DAC Output Routines 14
Split Axis Gantry Module Specific Routines 14
Miscellaneous Routines 14
Routines for Using the Setpoint Buffer 15
Routines for Reading/Writing Servo Parameters 15
Example Routines 16
Channel Numbers vs. Motor Numbers 17
Reading/Writing on a Motor-Number Basis 17
Reading/Writing on a Channel Basis 17
Parameter Descriptions 17
Initialization 18
Password Protection 18
Synchronous vs. Asynchronous Mode 19
Errors 20

2 Descriptions of Library Routines 21

3 Advanced Servo Library Example Programs 163

ex.motion( ) - Robot Motion Example 164
ex.servo.data( ) - Joint Motion Data Monitoring Example 168
ex.setpt( ) - Setpoint Buffer Example 172

ex.follow( ) - Axis Following Examples 178
ex.j4.gainset( ) - Dynamic Tuning of Adept Joint 4 Example 182
ex.calibrate( ) - Motor Calibration Example 185
ex.free( ) - FREE Mode Example 194

Index 197

List of Tables

Table 2-1 Advanced Servo Library Diagnostic Error Codes 24
Table 2-2 “Latched” Errors Bit Definition 55
Table 2-3 SPEC Servo Configuration Parameter Names 87
Table 2-4 SPEC Servo Data Parameter Names 92

1 Overview

Introduction 10
Compatibility 11
Adept Robot Access 11
Related Publications 12
Installation 12
Summary of Library Routines 12
Initialization and Other Utility Routines 12
Robot Motion Routines 13
Status Routines 13
Routines for Collecting Servo Data 13
Encoder and DAC Output Routines 14
Split Axis Gantry Module Specific Routines 14
Miscellaneous Routines 14
Routines for Using the Setpoint Buffer 15
Routines for Reading/Writing Servo Parameters 15
Example Routines 16
Channel Numbers vs. Motor Numbers 17
Reading/Writing on a Motor-Number Basis 17
Reading/Writing on a Channel Basis 17
Parameter Descriptions 17
Initialization 18
Password Protection 18
Synchronous vs. Asynchronous Mode 19
Errors 20

This chapter presents basic material and instructions for using the Advanced Servo Library.

Introduction

The Advanced Servo Library is a set of routines that allow you to perform robot control outside of the normal V^+ command set. These routines allow you to perform actions such as:

- Dynamic reading/writing of many SPEC parameters
- Direct control of a mechanism on an axis-by-axis basis
- Direct access to the setpoint buffer
- Direct access to joint servo data

The majority of V^+ applications will not require the Advanced Servo Library. These routines should be used only by programmers who are very familiar with the Adept control system. Normal V^+ operations include numerous checks to protect the mechanism. In many cases, the Advanced Servo Library routines bypass these checks and it becomes the responsibility of the programmer to ensure proper and safe mechanism motion.

Compatibility

The Advanced Servo Library is compatible with any AdeptMotion system, including Adept robots (limited to Joint 4 only) running V+ version 13.0 and having the Enhanced Trajectory Control License installed.

Adept Robot Access

Access to Adept robots is limited to Joint 4 only. Support for Adept robots is limited to the routine summaries listed in the following sections:

"Initialization and Other Utility Routines" on page 12

"Robot Motion Routines" on page 13

"Routines for Collecting Servo Data" on page 13

"Miscellaneous Routines" on page 14

"Routines for Reading/Writing Servo Parameters" on page 15

In addition, the routines **sl.version( )** and **sl.c.version( )** can be used with an Adept robot.

> **NOTE:** Any calls to routines outside of the sections indicated above will result in an error.

Related Publications

Before using these library routines you should be completely familiar with the contents of the *AdeptMotion VME Developer's Guide*, which describes the Adept control system and how it relates to installed mechanisms.

You should also be familiar with the V+ language as described in the *V+ Language User's Guide* and the *V+ Language Reference Guide*.

Installation

The Advanced Servo Library software is included on the Adept Utility Disk. The library routines compatible with V+ 13.0, which are protected and cannot be viewed or edited, are in the file ADVSERVO.LIB. The example routines, which are not protected and can be edited and copied, are in the file EXAMPLES.V2.

Summary of Library Routines

Following is a brief summary of the routines in the Advanced Servo Library. These routines are fully described in **"Descriptions of Library Routines" on page 21**.

Routines that have an **`sl.`** and **`sl.c.`** prefix can write to either a motor or a servo board channel. See **"Channel Numbers vs. Motor Numbers" on page 17**.

Initialization and Other Utility Routines

sl.init( ) Initialize data structures required by all other routines

sl.asl.cleanup( ) Delete the Advanced Servo Library global data structures

Robot Motion Routines

sl.enable( ) **sl.c.enable( )**	Enable/disable ASYNC Mode
sl.amp( ) **sl.c.amp( )**	Enable/disable Drive Enable signal
sl.calibrate( )	Initiate/monitor motor calibration
sl.drive( ) **sl.c.drive( )**	Move an axis a specified distance
sl.free( ) **sl.c.free( )**	Enable/disable FREE mode for a motor or motors
sl.stop( ) **sl.c.stop( )**	Halt a motion started with one of the drive routines
sl.brake( ) **sl.c.brake( )**	Release the specified axis brakes
sl.break( ) **sl.c.break( )**	Delay software execution until the last motion is complete
sl.mslew( )	Move one or more joints
sl.slew( )	Move to an encoder position every servo tick

Status Routines

sl.version( ) **sl.c.version( )**	Return various servo identification data
sl.status( ) **sl.c.status( )**	Read the status bits for an axis
sl.over.travel( )	Read the status of the over-travel switches
sl.read.opt.bit( )	Read the status of various system option bits

Routines for Collecting Servo Data

sl.listdata( )	Return status of servo data collection
sl.getdata( )	Retrieve the data collected from the servo task

sl.defdata( )	Define the data to be collected from the servo task
sl.deftrig( )	Define the data collection trigger mode and synchronization
sl.defwave( )	Define motion profile for axis testing
sl.wave.trig( )	Trigger motion profile defined with sl.defwave

Encoder and DAC Output Routines

sl.setdac( ) **sl.c.setdac( )**	Set the DAC to a specified voltage
sl.getdac( ) **sl.c.getdac( )**	Read the DAC output value
sl.adjust( ) **sl.c.adjust( )**	Adjust the position reference and mark the motor “calibrated”
sl.here( ) **sl.c.here( )**	Set the position reference and mark the motor “calibrated”
sl.where( ) **sl.c.where( )**	Read the current position of the motor

Split Axis Gantry Module Specific Routines

sl.set.gan.splt( )	Set split-axis gantry pointers
sl.slave.mode( )	Put motors into/out of slave mode

Miscellaneous Routines

sl.clr.ltch.err( )	Clear “Latched” errors word for specified motor
sl.jnt.to.mtr( )	Convert joint degrees or millimeters to encoder counts
sl.mtr.to.jnt( )	Convert encoder counts to joint degrees or millimeters
sl.pcocfg( ) **sl.c.pcocfg( )**	Configure Position Compare Output signal
sl.read.file( )	Read a SPEC data file into RAM
sl.wrt.dat.file( )	Write SPEC data to ASCII or binary disk file
sl.wrt.sys.file( )	Write all SPEC data to boot disk (for all robots)

Routines for Using the Setpoint Buffer

sl.setpt.cfg( )	Configure the setpoint buffer
sl.setpt.fill( )	Fill the setpoint buffer
sl.exttrig( )	Trigger the setpoint buffer

Routines for Reading/Writing Servo Parameters

sl.read( ) **sl.c.read( )**	Read a servo parameter
sl.write( ) **sl.c.write( )**	Write a servo parameter
sl.rw.accel( )	Read/write Cartesian acceleration
sl.rw.amask( )	Read/write the axis enable mask
sl.rw.bcklsh( )	Read/write backlash compensation
sl.rw.calcfg( )	Read/write calibration configuration
sl.rw.cal.file( )	Read/write robot calibration program file
sl.rw.collision( )	Read/write robot collision avoidance offsets
sl.rw.colsn.det( )	Read/write static obstacle detection parameters
sl.rw.comp( )	Read/write linear compensation table
sl.rw.cspd( )	Read/write Cartesian speed
sl.rw.ctol( )	Read/write coarse tolerance
sl.rw.enccfg( )	Read/write encoder configuration
sl.rw.ftol( )	Read/write fine tolerance
sl.rw.hand( )	Read/write hand control signals
sl.rw.hdw.latch( )	Read/write axis-specific encoder latching configuration
sl.rw.id.data( )	Read/write robot name, option word, model and serial numbers
sl.rw.jaccel( )	Read/write joint acceleration

sl.rw.jmap( ) Read/write motor mapping for robot or external encoders

sl.rw.link( ) Read/write robot link dimensions

sl.rw.maxcspd( ) Read/write maximum Cartesian speed

sl.rw.maxjspd( ) Read/write maximum joint speed

sl.rw.max.c.acc( ) Read/write maximum Cartesian acceleration

sl.rw.max.j.acc( ) Read/write maximum joint acceleration

sl.rw.mspd( ) Read/write maximum manual joint speed for MCP

sl.rw.mtic( ) Read/write smallest joint increment using MCP

sl.rw.njts( ) Read/write the number of joints for the selected robot

sl.rw.pwr.decel( ) Read/write arm power off deceleration

sl.rw.rollover( ) Read/write encoder roll-over value

sl.rw.scale( ) Read/write encoder scale factors

sl.rw.softstops( ) Read/write working and maximum soft stop limits

sl.rw.s_crv( ) Read/write S-curve profiles

sl.rw.tooloff( ) Read/write robot tool offset

sl.rw.tuning( ) Read/write any or all of the tuning parameters for a motor

Example Routines

ex.motion( ) - Robot Motion Example

ex.setpt( ) - Setpoint Buffer Example

ex.follow( ) - Axis Following Examples

ex.servo.data( ) - Joint Motion Data Monitoring Example

ex.j4.gainset( ) - Dynamic Tuning of Adept Joint 4 Example

ex.calibrate( ) - Motor Calibration Example

ex.free( ) - FREE Mode Example

Channel Numbers vs. Motor Numbers

There are two ways of selecting a servo channel to write to/read from. The first is to specify a robot motor number; the second is to specify a motion board number and a channel number on that board.

Reading/Writing on a Motor-Number Basis

Routines that read/write on a motor-number basis have a parameter for the motor number. The motor number is determined by the axis-to-motor mapping that is specified in the device module for each robot. The motor number will always be associated with the currently SELECTed robot.

> **NOTE:** In most cases, a motor number and a joint number will be the same. However, in cases such as a robot with split-axis control, this is not the case and you will have to refer to the device module documentation for the correct motor number assignments.

In some cases, the read/write routines refer to values that apply to all motors in a kinematic description. In these cases, the value is associated with the currently SELECTed robot.

Reading/Writing on a Channel Basis

Routines that read/write on a channel basis have parameters for a board number and a channel number. These routines read/write the specified board/channel without regard for the motor assignments in the currently selected robot's device module description. Board numbers are based on the physical settings of the address DIP switches on the motion board. See the ***AdeptMotion VME Developer's Guide*** for details on setting these DIP switches. See the ***AdeptMotion VME Developer's Guide*** for details on the physical connections that correspond to each channel.

Parameter Descriptions

The Advanced Servo Library routines allow you to read and write parameters that are normally accessible only with the SPEC utility program. This manual does not give a complete description of each parameter's use and interactions. See the ***AdeptMotion VME Developer's Guide*** for a complete description of all parameters.

Initialization

Before any routines in the Advanced Servo Library can be used, the routine **sl.init( )** must be run. This routine initializes all of the global data structures that are required by the routines described in **"Descriptions of Library Routines" on page 21**.

When this routine is selected, the system checks for the SPECSUB library to be in system memory. If it is not in memory, the program attempts to load the file SPECSUB.LIB from the same path from which the program was originally loaded. If the file cannot be found, the user is asked to enter the appropriate path specifications to access the file. No routine in the Advanced Servo Library can be used until the SPECSUB.LIB file has been loaded, and the **sl.init( )** routine has successfully initialized the global data structures.

When you are done using the Advanced Servo Library, the routine **sl.asl.cleanup( )** can be called to delete these data structures and free up the allocated memory.

The command:

```
DELETEM a.advservo
```

will delete all the advanced servo routines from memory.

Password Protection

Any robot that has a password specified in its SPEC file will also be protected against **write** operations of the Advanced Servo Library. **Read-only** access is allowed for all robots. Password protection for the Advanced Servo Library is done on a mechanism-specific basis, not a parameter-by-parameter basis. See the description of **sl.init( )** for details on entering passwords.

Synchronous vs. Asynchronous Mode

Normally, the Adept servo task expects commands from V+once each V+ trajectory cycle. This synchronous mode allows normal control of a mechanism using the standard MOVE, MOVES, etc., instructions. However, many of the routines in the Advanced Servo Library require the asynchronous mode, which allows servo commands that are not generated by normal V+ commands. In order to use the routines that require asynchronous mode, you must first place the desired servo channels in asynchronous mode using the **sl.enable( )** or **sl.c.enable( )** routines. Asynchronous mode tells V+ that its commands are being ignored and that commands issued directly from a user program are being processed.

When a channel is in asynchronous mode, normal V+ motion instructions will be ignored. Thus, when you have completed any asynchronous operations, you must again call **sl.enable( )** or **sl.c.enable( )** to disable asynchronous mode before using normal V+ instructions. An alternative method of disabling asynchronous mode is to disable high power. Subsequent power-up of the robot will then reenable synchronous mode automatically.

Errors

In general, the Advanced Servo Library routines return error values in the normal V+ ranges:

0 or 1 Success
<0 Error as described in the ***V+ Language Reference Guide***

The following are special errors defined for the Advanced Servo Library:

–1 No system data to edit for the referenced robot. That is, the data is irrelevant for this type of robot (no data is read or written).

–2 Internal consistency error. Make sure all referenced motors and boards are installed and configured (no data is read or written).

–3 Entered value too small (write operation only).

–4 Entered value too large (write operation only).

–5 Illegal digital I/O number entered (write operation only).

–6 Illegal data type entered, e.g., a string entered where a numeric value is required.

These values are always returned by the called routine. In order to provide maximum flexibility for user interfaces, error messages are displayed on the screen only when the variable **sl.adv.diag** has been set to TRUE, or the Advanced Servo Library routine **sl.adv.diag( )** is executed.

In some cases, if a read operation is successful, the status output variable will contain a positive number that conveys quantitative information about the results of the operation.

2 Descriptions of Library Routines

This chapter describes the functions and calling sequences of the routines in the Advanced Servo Library. These routines may be called by application software written by a system customizer.

The descriptions of the routines are presented in alphabetical order, with each routine starting on a separate page. The description for each routine contains the following sections, as applicable.

Calling Sequence

The format of a V+ CALL instruction for the routine is shown.

> **NOTE:** The variable names used for the routine parameters are for explanation purposes only. Your application program can use any variable names you want when calling the routine.
> Although some CALL instructions are shown continued on a second line, instructions in an actual V+ program must always be completed on one line.

Function

This is a brief statement of the function of the routine.

Usage Considerations

This section is used to point out any special considerations associated with use of the routine.

Input Parameters

Each of the input parameters in the calling sequence is described in detail. For any parameter that has a restriction on its acceptable values, or is required to be defined as a double-precision variable, the restriction/requirement is specified.

Output Parameters

Each of the output parameters in the calling sequence is described in detail.

Details

A complete description of the routine and its use is given.

Related Routines

Other library routines with related functions are listed.

Calling Sequence

```
CALL sl.adjust (mot.num, count, error)
```

Function

This program incrementally adjusts the axis position reference (both commanded and reported actual position) by the amount specified and marks the motor "calibrated" without moving the motor.

Usage Consideration

The indicated motor must be in ASYNC mode.

Input Parameters

`mot.num`	Motor number
`count`	Value, in encoder counts, to add to the position reference

Output Parameter

`error`	Standard success/error code

Details

WARNING: Any existing calibration will be invalidated, but the robot will still be considered to be calibrated and able to move.

Related Routines

sl.c.adjust
sl.c.here
sl.here

Calling Sequence

```
CALL sl.adv.diag (error)
```

Function

This is a service routine that types error messages to the MONITOR window.

Input Parameter

`error` Standard V+ error code or special Advanced Servo Library diagnostic error code.

Output Parameters

None

Details

This routine is called automatically by the **sl.rw.*( )** routines if the global variable **sl.adv.diag** is nonzero.

The special Advanced Servo Library diagnostic error codes are returned by the low level routines that read/write the SPEC parameters. These codes represent the conditions described in **Table 2-1**.

Table 2-1. Advanced Servo Library Diagnostic Error Codes

Error Code	Description
–1	No system data to edit for the referenced robot, i.e., the data is irrelevant for this type of robot (no data was read or written).
–2	Internal consistency error; make sure all referenced motors and boards are installed and configured (no data was read or written).
–3	Value too small (write operation only).
–4	Value too large (write operation only).
–5	Illegal digital I/O number (write operation only).
–6	Illegal data type, i.e., a string.

Calling Sequence

```
CALL sl.amp (mot.num, enable, error)
```

Function

This program turns on or off the Drive Enable signal.

Usage Considerations

The indicated motor must be in ASYNC mode.

Input Parameters

`mot.num`	Motor number
`enable`	TRUE to turn ON the Drive Enable signal FALSE to turn OFF the Drive Enable signal

Output Parameter

`error`	Standard success/error code

Details

This program enables or disables the Drive Enable signal for the specified motor. In order to command motion of the motor, its amplifier must be enabled by the Drive Enable signal.

Related Routine

sl.c.amp

Calling Sequence

```
CALL sl.asl.cleanup ()
```

Function:

Deletes all the global data structures defined by the Advanced Servo Library.

Input Parameters

None

Output Parameters

None

Details

This routine may be called whenever you are finished using the Advanced Servo Library. It deletes all the global data structures from memory.
The command:

```
DELETEM a.advservo
```

will delete all the Advanced Servo Library routines.

Calling Sequence

```
CALL sl.brake (mot.num, enable, error)
```

Function

Turn on or off the axis brake.

Usage Considerations

The indicated motor must be in ASYNC mode.

Input Parameters

`mot.num`	Motor number
`enable`	If TRUE, enable the brake. If FALSE, disable the brake.

Output Parameter

`error`	Standard success/error code

Details

This routine will enable or disable the brake signal for the specified motor.

WARNING: The axis brake should not be enabled for joints that are under servo control. Be very careful when releasing the brake on a gravity-loaded joint that is not under servo control because the joint may drop immediately.

Related Routine

sl.c.brake

Calling Sequence

```
CALL sl.break (mot.num, error)
```

Function

This program checks the device status information to assure that the most recent motion command for the specified motor has completed before proceeding.

Input Parameter

`mot.num` Motor number

Output Parameter

`error` Standard success/error code

Details

This routine performs a motor-specific function similar to the V+ BREAK instruction. Program execution will be delayed until the motor is within the tolerance (COARSE, FINE) specified by the application program.

Related Routine

sl.c.break

Calling Sequence

```
CALL sl.c.adjust (board, channel, count, error)
```

Function

This program incrementally adjusts the axis position reference (both commanded and reported actual position) by the amount specified and marks the motor "calibrated" without moving the motor.

Usage Considerations

The indicated channel must be in ASYNC mode.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**
`count`	Value, in encoder counts, to add to the position reference

Output Parameter

`error`	Standard success/error code

Details

WARNING: Any existing calibration will be invalidated, but the robot will be considered calibrated and able to move.

Related Routines

sl.adjust
sl.c.here
sl.here

Calling Sequence

```
CALL sl.c.amp (board, channel, enable, error)
```

Function

This program turns on or off the Drive Enable signal for the indicated channel.

Usage Considerations

The indicated channel must be in ASYNC mode.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**
`enable`	TRUE to turn ON the drive enable signal FALSE to turn OFF the drive enable signal

Output Parameter

`error`	Standard success/error code

Details

This program enables or disables the Drive Enable signal for the specified channel. In order to command motion of the motor, its amplifier must be enabled by the Drive Enable signal.

Related Routine

sl.amp

Calling Sequence

```
CALL sl.c.brake (board, channel, enable, error)
```

Function

This program turns the axis brake on or off.

Usage Considerations

The indicated channel must be in ASYNC mode.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**
`enable`	If TRUE, enable the brake. If FALSE, disable the brake.

Output Parameter

`error`	Standard success/error code

Details

This routine will enable or disable the brake signal for the specified channel. Typically, the brakes are *released* when this signal is enabled.

WARNING: The axis brake should not be enabled for joints that are under servo control. Be very careful when releasing the brake on a gravity loaded joint that is not under servo control because the joint may drop immediately.

Related Routine

sl.brake

Calling Sequence

```
CALL sl.c.break (board, channel, error)
```

Function

This program checks the device status information to assure the axis is stabilized or in tolerance before proceeding.

Usage Considerations

The indicated channel must be in ASYNC mode.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**

Output Parameter

`error`	Standard success/error code

Details

This routine performs a channel-specific function similar to the V+ BREAK instruction. Program execution will be delayed until the motor is within the tolerance (COARSE, FINE) specified by the application program .

Related Routine

sl.break

Calling Sequence

```
CALL sl.c.drive (board, channel, accel, spd, dist, abs,
                                          error)
```

Function

This program moves the motor a specified distance. Acceleration and speed are included to control the motion.

Usage Considerationss

The program does not check for encoder roll-over.
The indicated channel must be in ASYNC mode.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**
`accel`	Amount of time to reach max speed (in milliseconds)
`spd`	Maximum speed in encoder counts/millisecond
`dist`	The distance the axis is to move (in encoder counts)
`abs`	If TRUE, drive to the actual position If FALSE, drive to relative position

Output Parameter

`error`	Standard success/error code

Details

This routine performs an operation similar to the V+ DRIVE program instruction. However, the programmer has the option to make the move to an absolute position or relative to the current position of the motor.

Related Routines

sl.drive
sl.mslew
sl.slew

Calling Sequence

```
CALL sl.c.enable (board, channel, enable, finish, error)
```

Function

This program enables/disables asynchronous mode for controlling a motion board channel.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**
`enable`	If TRUE, enable ASYNC mode for specified **channel** If FALSE, disable ASYNC mode for specified **channel**
`finish`	If TRUE, the program will wait until the last motion instruction for the indicated **channel** has completed before DETACHing and ATTACHing the robot.

Output Parameter

`error`	Standard success/error code

Details

Before any routines in the Advanced Servo Library that require ASYNC mode can function, ASYNC mode for any referenced channel must be set. Once ASYNC mode is set, normal V+ motion instructions will be ignored. To resume normal operation, ASYNC mode must be disabled. ASYNC mode is enabled/disabled on a channel-by-channel basis.

While a motor or motors are in ASYNC mode, V+ will continue to plan a motion for the motor(s), however, motion control is left to the application program using the Advanced Servo Library routines. The remaining motors on the robot will continue to function normally. That is, the robot can be commanded to move using the standard V+ motion commands.

> **NOTE:** ASYNC mode can also be disabled by issuing the V+ command DISABLE POWER. Subsequent power-up of the robot will enable SYNC mode.

Related Routine

sl.enable

Calling Sequence

```
CALL sl.c.free (board, channel, enable, error)
```

Function

Places the desired channel into FREE mode, if FREE mode is allowed for the motor attached to the channel.

Usage Considerations

This routine will automatically place the required channel into and out of ASYNC mode.

The "FREE mode digital signal" defined in SPEC is supported by this routine.

If disabling FREE mode, the channel will automatically be placed back into SYNC mode (position servo mode) and the robot will be DETACHed and reATTACHed to the calling task.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**
`enable`	If TRUE, enable FREE mode for specified **channel** If FALSE, disable FREE mode for specified **channel**

Output Parameter

`error`	Standard success/error code. If FREE mode is not allowed (it is disabled in SPEC) for a requested motor, error code –609 (`*Illegal joint number*`) is returned. An informational message is typed to the monitor if the global flag **sl.adv.diag** is nonzero.

Details

WARNING: Placing a gravity-loaded joint into FREE mode could cause the joint to drop causing damage to the payload or the workspace.

FREE mode releases servo control of the motor by either disabling the Drive Enable signal for the specified channel, or by commanding zero torque. The specific method depends on the drive configuration specified in SPEC.

This mode is intended to allow the mechanism to be easily moved by hand during setup, or to create a compliant robot joint or joints to ease part insertion.

This routine places only one channel into or out of FREE mode. To place multiple motors into FREE mode, use **sl.free( )**.

Related Routine

sl.free

Calling Sequence

```
CALL sl.c.getdac (board, channel, select, dac, error)
```

Function

Returns the current DAC value being output to the selected channel.

Usage Considerations

The indicated channel must be in ASYNC mode.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**
`select`	If TRUE, **dac** is returned as a voltage (±10VDC) If FALSE, **dac** is returned in DAC units (±32,767)

Output Parameters

`dac`	Current DAC output (in DAC counts or volts depending on the value of **select**)
`error`	Standard success/error code

Details

This function returns the current value of the DAC output to the motor amplifiers. Since there normally is always some steady-state holding torque applied when a joint is servoing in place, this value will generally be 0 only when arm power is disabled or the motor is uncalibrated. (In the case of velocity-style drives, 0 indicates no commanded velocity.)

Related Routines

sl.getdac
sl.setdac
sl.c.setdac

Calling Sequence

```
CALL sl.c.here (board, channel, count, error)
```

Function

This program sets the encoder register to the indicated value and marks the motor "calibrated" without moving the motor.

Usage Considerations

The indicated channel must be in ASYNC mode.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**
`count`	Encoder value to place in the register

Output Parameter

`error`	Standard success/error code

Details

This routine sets the position reference of the motor attached to the indicated **channel**. This is useful, for example, in custom calibration programs that require reading a resolver or absolute encoder to get the position without moving the motor. After computing the correct position in encoder counts, this routine can be called to set the position reference.

WARNING: Any existing calibration will be invalidated, but the robot will be considered to be calibrated and able to move.

Related Routines

sl.here
sl.adjust
sl.c.adjust
sl.c.where
sl.where

Calling Sequence

```
CALL sl.c.pcocfg (board, channel, pco.st, pco.pos, delta,
                                    equ, frm.servo[], error)
```

Function

This program configures the Position Compare feature in the servo code. Full support is available only on VMI based motion systems. There is reduced functionality on the EJI belt encoder channels 7 & 8. It is NOT supported on MI-3/MI-6 motion boards.

Input Parameters

`board` Board number

`channel` Channel number on **board**

`pco.st` Initial PCO State (VMI ONLY)
< 0 - No change
= 0 - Clear PCO
> 0 - Set PCO

`pco.pos` Initial compare position, in encoder counts

`delta` New compare delta, in encoder counts

`equ` New EQUal signal configuration. Bit mask as follows ("xx" means not important)

Binary Bitmask	Hex Bitmask	PCO Operation
00xx	^H0	Nothing. Physical PCO directly controllable with **pco.st**. (VMI ONLY)
0100	^H4	PCO pulses when motor position matches the PCO position **pco.pos**
0101	^H5	PCO toggles when motor position matches the PCO position **pco.pos**
0110	^H6	PCO is deasserted when motor position matches the PCO position **pco.pos**

Binary Bitmask	Hex Bitmask	PCO Operation
0111	^H7	PCO is asserted when motor position matches the PCO position **pco.pos**
10xx	^H8	Fire Camera Strobe #0 (CSTRB0) when motor position matches the PCO position **pco.pos**
11xx	^HC	Fire Camera Strobe #1 (CSTRB1) when motor position matches the PCO position **pco.pos**

Output Parameters

`frm.servo[ ]` Real array defined as follows:
[0] = Previous state of PCO output for this motion board
= 0 PCO was cleared
> 0 PCO was set
[1] = Previous compare position in encoder counts for this axis
[2] = Previous compare delta, in encoder counts
[3] = Previous compare configuration for this axis

`error` Success/error code. If successful, contains number of elements in the **frm.servo[ ]** array.

Details

The Position Compare Output (PCO) function generates an internal signal called "Equal" (EQU) when the actual position of the axis exactly matches the target position. This signal may be routed to either the physical PCO (on the Machine connector on the VMI ONLY), or one of the backplane Camera Strobe lines (CSTRB0 and CSTRB1). After a trigger is generated, the Position Compare function will automatically update the target position by adding the input parameter **delta** to it. (See Note #3).

On the VMI, the input argument **pco.st** controls the initial state of the physical PCO signal. That is, when this routine is called, the PCO signal will be set to the state indicated by the **pco.st** argument. If the application requires that the PCO signal be asserted when the axis passes over the trigger position (**pco.pos**), make sure that the initial PCO state is cleared when this routine is called (**pco.st** = 0).

NOTE:

1. On the VMI, each channel produces an EQU signal, but there is only one physical PCO per board, and no matter how it is configured, the PCO will always be directly controlled with the **pco.st** argument. The PCO tracks the most recent change in state of any of its available trigger sources (position compare triggers or **pco.st**). If two sources change at exactly the same time, clearing the PCO signal has priority over asserting the PCO signal.
2. V^+ can use the PCO function to generate camera strobes. Changing this configuration on a belt encoder may therefore interfere with the operation of the camera strobes.
3. The compare delta is unidirectional, i.e., after the initial compare position causes a match (a PCO trigger), the input parameter **delta** is added to it to form a new compare position. The old compare position will not trigger any changes in the PCO or camera strobe lines should the axis back up over it again.

   To force the system to trigger using the old position, simply call this routine again to reconfigure the PCO before backing over the position.
4. The EJI supports the Position Compare function on its belt encoder channels only, and has no physical PCO output. Several of the options described above are not available on its belt channels. The functions labeled "VMI ONLY" do not apply for the EJI belt channels.

Related Routine

sl.pcocfg

Calling Sequence

```
CALL sl.c.read (board, channel, opcode, frm.servo, error)
```

Function

This routine reads motion parameter values from the servos or V^+.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**
`opcode`	Symbolic name of the parameter to read (see the description of **sl.read( )** for a list of the SPEC parameters that can be read)

Output Parameters

`frm.servo`	If successful, contains the requested parameter value
`error`	Standard success/error code

Details

This routine reads the SPEC parameters listed in the **sl.read( )** routine description. If the global variable **sl.adv.diag** is nonzero, error messages will be displayed on the monitor.

Related Routines

sl.rw.*
sl.read
sl.c.write
sl.write

Calling Sequence

```
CALL sl.c.setdac (board, channel, select, value, error)
```

Function

This program sets the DAC for a specified channel to a specified output.

Usage Considerations

The maximum output voltage is ±10 volts. The maximum DAC value is ±32,767.

The indicated channel must be in ASYNC mode.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**
`select`	(Optional) If TRUE, **value** contains a voltage (±10). If FALSE, **value** contains DAC counts (±32,767). Default is FALSE.
`value`	Desired analog voltage output, or DAC value to set. If undefined, the DAC is set to 0.

Output Parameter

`error`	Success/error code

Details

WARNING: This function could cause serious damage to the mechanism, tooling, workspace, or personnel. Executing this function with arm power on may cause the axis specified to move at VERY HIGH SPEED. This command is **open loop: USE EXTREME CAUTION** when running this program.

Related Routine

sl.setdac

Calling Sequence

```
CALL sl.c.status (board, channel, c.status, error)
```

Function

This program reads the status bits from the servo database for the specified board and channel.

Usage Considerations

The output parameter **c.status** must be declared as a DOUBLE or AUTO DOUBLE by the calling program.

Input Parameters

`board` Board number

`channel` Channel number on **board**

Output Parameters

`c.status` Status word for this channel

`error` Standard success/error code

Details

The status word bits are interpreted as follows (undefined bits are reserved and do not return useful information):

Bit	Hex Bitmask	Description
0	^H1	Home Switch state for this axis (input)
1	^H2	Spare Input state for this axis (input)
8	^H100	Spare Output state for this axis (output)
9	^H200	Brake release status for this axis (output)
10	^H400	Drive/motor enable status for this axis (output)
12	^H1000	High Power Enable state, all drives (output)
13	^H2000	Set if Hand solenoids are “active”

Bit	Hex Bitmask	Description
16	^H1 0000	"Multiple cycle" command complete (such as **sl.drive( )**)
17	^H2 0000	Set if motor is capable of servoing
24	^H100 0000	Motor is in tolerance
25	^H200 0000	First zero index seen after RESET or CAL
26	^H400 0000	Stop-on-force detected
30	^H4000 0000	Set if High Power is enabled and drive is enabled
31	^H8000 0000	Motor has been calibrated

Related Routine

sl.status

Calling Sequence

```
CALL sl.c.stop (board, channel, error)
```

Function

Stop a motion started with a call to the routine **sl.drive( )** or **sl.c.drive( )**.

Usage Considerations

The indicated channel must be in ASYNC mode.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**

Output Parameter

`error`	Standard success/error code

Details

This routine stops a motor using the normal deceleration phases of a motion (this is not an immediate halt of motor motion).

Related Routines

sl.drive
sl.c.drive
sl.stop

Calling Sequence

```
CALL sl.c.version (board, channel, $version[], version[],
                                   error)
```

Function

This program returns various servo identification, as well as basic information on the selected joint.

Input Parameter

`board` Board number

`channel` Channel number on **board**

Output Parameters

`$version[ ]` Array of strings as follows:
[0] = Servo Code version, revision, edit, and fork number
(Format: vvref Example: 122B1)

`version[ ]` Real array containing the following:

- [0] Number of channels on **board**
- [1] Axis type (1 = primary, 2 = secondary)
- [2] Servo loop rate in Hz
- [3] Max DAC for this motor
- [4] Servo Board Type:
  - 0 = VMI
  - 1 = ACB
  - 2 = VJI
  - 3 = VFI
  - 4 = MI-6 or MI-3
- [5] Robot number that owns joint mapped to **board** and **channel**
- [6] Motor number corresponding to joint mapped to **board** and **channel**
- [7] Board number. Matches **board** input argument.
- [8] Channel number. Matches **channel** input argument.
- [9] Processor # running servo code for joint mapped to **board** and **channel**.
- [10] Calibration compatibility code
- [11] Robot serial number
- [12] Servo mechanism ID
- [13] Servo board revision

error Standard success/error code

Related Routine

sl.version

Calling Sequence

```
CALL sl.c.where (board, channel, cts, loc, error)
```

Function

Reads current location of the motor mapped to **board** and **channel**.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**

Output Parameters

`cts`	Current position in encoder counts
`loc`	(Optional) Current position in millimeters or degrees
`error`	Standard success/error code

Details

This routine returns the current position of the motor mapped to the **board** and **channel** input arguments. It always returns the current position in encoder counts. If the **loc** argument is specified, the routine will also compute the current location in joint units (millimeters or degrees).

Related Routines

sl.where
sl.c.here
sl.here

Calling Sequence

```
CALL sl.c.write (board, channel, opcode, value, error)
```

Function

This program writes tuning and other SPEC parameters to V+ or the servos (in RAM only).

Usage Considerations

Since motion parameters are saved based on a kinematic description, parameters changed on a board/channel basis will not be saved when a new SPEC file is created or parameters are saved to the system file. The routine **sl.write( )** must be used if you want to permanently save edited motion parameters to the system disk using the routine **sl.wrt.sys.file( )**.

Input Parameters

`board`	Board number
`channel`	Channel number on **board**
`opcode`	Symbolic name of desired value (see the description of **sl.read( )** for a list of the SPEC parameters that can be written)
`value`	New value for the specified parameter

Output Parameter

`error`	Standard success/error code

Details

The value for the motion parameter will be changed in local RAM only. The value is not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routines

sl.rw.*
sl.read
sl.c.read
sl.write

Calling Sequence

```
CALL sl.calibrate (mot.num, srch.dist, spd.max, accel.time,
                                    abs, srch.type, hard.tol,
                                    states[], error)
```

Function

Performs trajectory generation and home-switch latching associated with calibration.

Usage Considerations

The indicated motor must be in ASYNC mode.

Input Parameters

`mot.num` Motor number

`srch.dist` (Optional) search distance in encoder counts. If undefined, NULL TRAJECTORY is assumed.

`spd.max` (Optional) maximum speed (unsigned) in counts/millisecond. If undefined, default speed data is assumed.

`accel.time` (Optional) acceleration time in msec. If undefined, default acceleration data is assumed.

`abs` (Optional) Absolute/Relative motion flag
TRUE for Absolute motion
FALSE for Relative motion

`srch.type` (Optional) Search Type:
–1 Don't stop at transition
0 Stop at Home Switch transition
+1 Stop at Hard Stop transition

`hard.tol` (Optional) Used ONLY if **srch.type** = 1.
Specifies the position tolerance to use when searching for the hard stop. This parameter is ignored for all other search types.

Output Parameters

`states[ ]` Current state of command as follows:
[0] Current state of trajectory
3 = Accel
2 = Constant Velocity
1 = Decel

0 = Done
[1] Current state of Home Switch
[2] Number of calibration features or transitions last seen. Four optional outputs, depending on how many features, or transitions, were seen:
[3] Position of 1st calibration feature
[4] Position of 2nd calibration feature
[5] Position of 3rd calibration feature
[6] Position of 4th calibration feature

`error` Standard success/error code. If successful, contains number of elements in the **states[ ]** array.

Details

This routine is meant to assist a V+ program as it performs the sequence of operations necessary to calibrate a robot. Each time this routine is called, the "count of calibration feature transitions" will be zeroed and the previous latched positions will be cleared. If a non-null trajectory is passed in, the axis will be placed in "calibration mode", latching of home switch transitions will be enabled and the indicated motion (if any) will be started.

If the speed or accel time is 0, then the defaults (data in opcodes **sl.trjdvm** and **sl.trjdta**, respectively) will be used.

If the distance is 0, or if no trajectory parameters are passed in, a null trajectory is assumed. This means the home latching logic will be reset. A NULL TRAJECTORY WILL NOT MODIFY THE TRAJECTORY OF ANY PREVIOUS COMMANDS.

If a positive search configuration is passed in (that is, the input parameters are defined), the motion will be halted if the search for the home switch or hard stop was successful.

During the motion, the following calibration-specific operations will be performed:

1. The timeout on `*Motor Stalled*` errors will be set to the value in **sl.homsat** (Opcode 1044).
2. Calibration feature transitions that occur during the motion will be latched into an array for later reading. Up to four previous values will be latched.

Upon completion of the motion, "calibration mode" will be cleared, and latching of home positions will cease.

NOTE: This means that calibration features that occur close to the destination may not be returned (because the actual position may lag the commanded trajectory, and latching is disabled when the commanded trajectory is complete).

A V+ program would typically:

1. Call this routine with a defined trajectory to start the search for the calibration feature (object: home switch, hard stop).
2. Poll for motion completion or latched home switch transitions (or hard stop position) by repeatedly calling this routine with a NULL trajectory (i.e., no inputs except motor number).
3. Determine a position reference (perhaps by moving to the next zero index after the home switch).
4. Use the **sl.adjust( )** or **sl.here( )** routines to set the new position reference and mark the axis as calibrated.

See the example program **ex.calibrate( ) - Motor Calibration Example** for details.

Related Routine

sl.rw.calcfg

Calling Sequence

```
CALL sl.clr.ltch.err (mot.num, error)
```

Function

This program clears the "Latched" errors word in the servo code for the specified motor.

Input Parameter

`mot.num` Motor number. To clear the "Latched" errors for ALL motors attached to the currently SELECTed robot, set this value to 0.

Output Parameter

`error` Standard success/error code.

Details

The "Latched" errors word is used to report errors that are of a transient nature but must not be missed by V+. When the robot is known to be in a safe state (e.g., arm power is disabled), V+ clears this word after reporting any errors. The value of the "Latched" errors word can be read using the **sl.read( )** routine to read the value of opcode **sl.latch**.

The "Latched" errors word is defined as follows:

Table 2-2. "Latched" Errors Bit Definition

Bit	V+ Error Message (V+ error code)	Description
0	`*RSC communications failure* (-651)`	V+ has lost communications with the robot signature card (RSC).
1	`*RSC hardware failure* (-669)`	The RSC has reported an internal problem.
2	`*RSC power failure* (-670)`	The RSC has reported that its power is failing.
3	`*Servo board 12v fuse open* (-671)`	Your servo board has a fused 12v bus, and the system has detected an open circuit at that location.

Table 2-2. "Latched" Errors Bit Definition (Continued)

Bit	V+ Error Message (V+ error code)	Description
4	*Servo board solenoid fuse open* (–672)	Your servo board has a fused robot solenoid control line, and the system has detected an open circuit at that location.
5	*Servo board E-STOP fuse open* (–673)	Your servo board has a fused ESTOP circuit, and the system has detected an open circuit at that location.
6	*E-STOP from robot* (–640)	The motion interface board (VJI) has detected an E-STOP condition generated by the RSC in the robot. (Adept robots only)
7	*E-STOP from amplifier* (–641)	The motion interface board has detected an E-STOP condition generated by the amplifiers.
8	*Motor amplifier fault* Mtr n (–1018)	The power amplifier for the indicated motor has signaled a fault condition on fault line 1. This fault occurs only for devices controlled by an AdeptMotion Servo system.
9	*No air pressure* (–607)	V+ detected that the air supply to the robot brakes and hand has failed. (Adept robots only)
10	*Motor overheating* Mtr n (–1016)	The indicated motor is overheating
11	*Duty-cycle exceeded* Mtr n (–1021)	The indicated motor has been driven fast for too long a period of time.
12	*E-STOP from backplane* (–643)	The motion interface board has detected an E-STOP due to the BRAKE-ESTOP signal being asserted on the VMEbus.
13	*E-STOP from SYSFAIL* (–642)	The motion interface board has detected an E-STOP due to the SYS-FAIL signal being asserted on the VME bus.

Table 2-2. "Latched" Errors Bit Definition (Continued)

Bit	V+ Error Message (V+ error code)	Description
14	`*Negative overtravel* Mtr n* (-1032)`	The indicated motor has moved beyond the hardware-limited negative range of motion.
15	`*Positive overtravel* Mtr n (-1033)`	The indicated motor has moved beyond the hardware-limited positive range of motion.
16	`*Motor stalled* Mtr n (-1007)`	The indicated motor has stalled while being driven.
17	`*Envelope error* Mtr n (-1006)`	The indicated motor was not tracking the commanded position with sufficient accuracy, indicating a failure in the hardware servo system or something impeding the path of the robot.
18	`*Unexpected zero index* Mtr n (-1005)`	A zero index signal was received from the encoder for this motor at an unexpected time.
19	`*Encoder quadrature error* Mtr n (-1008)`	The encoder from the specified motor is sending information that is not phased correctly.
20	`*No zero index* Mtr n (-1004)`	The motion system did not detect a zero-index mark for the indicated joint.
21	`*Hard envelope error* Mtr n (-1027)`	The indicated motor was not tracking the commanded position with sufficient accuracy, indicating a failure in the hardware servo system or something impeding the path of the robot.
22	`*Encoder fault* Mtr n (-1025)`	The servo board has detected a broken encoder wire on the indicated axis.
23	`*Skew envelope error* Mtr n (-1022)`	The two motors associated with a split axis were not tracking each other with sufficient accuracy.

Table 2-2. "Latched" Errors Bit Definition (Continued)

Bit	V+ Error Message (V+ error code)	Description
24	`*Manual brake release* (-639)`	The robot's manual brake-release button is active.
25	`Unused`	
26	`Unused`	
27	`Unused`	
28	`*Servo task overloaded* (-674)`	A servo interrupt task has used up all the execution time. The detection algorithm reports an error when the servo interrupt task completely occupies 10 or more time slices per second of real time. The robot went to a fatal error state when this error occurred, and the servo interrupt task stopped running.
29	`*Force protect limit exceeded* (-624)`	At least one force-sensor strain gauge reading has exceeded the preset limit, causing a robot panic stop.
30	`*Force sensor communication error (-634)`	Force communication error has occured.

Related Routine

sl.read

Calling Sequence

```
CALL sl.defdata (mot.num, coll.time, up.cycle, n.args,
                                        user.ind[], error)
```

Function

This program selects what type of data is to be collected during a robot motion.

Input Parameters

`mot.num`	Motor number
`coll.time`	Amount of time to collect data, in milliseconds
`up.cycle`	Cycles between updates (0 = max rate)
`n.args`	Number of data items to be collected (clipped at 8)
`user.ind[ ]`	User's index of data to be collected; see "Details"

Output Parameters

`coll.time`	Actual time to collect data, in milliseconds
`up.cycle`	Actual time between updates, in milliseconds
`error`	Standard success/error code

Details

This routine, along with **sl.getdata( )** and **sl.defwave( )** allows you to see all the servo statistics that are generated when a robot motion is made. See the example program **ex.servo.data( ) - Joint Motion Data Monitoring Example** for details.

The following global variables can be used to specify the type of data to collect:

sl.caltrq	Torque commanded to drive
sl.cmdvel	Commanded velocity
sl.velerr	Velocity error
sl.motvel	Actual (motor) velocity
sl.cmdpos	Commanded position (setpoint)
sl.poserr	Position error
sl.motpos	Actual (motor) position
sl.eplpos	Last latched position of encoder
sl.inxpos	Last position of zero index
sl.inxdel	Actual count between zero indexes
sl.inxerr	Running count of index errors

sl.interr	Integrated position error
sl.ganskw	Split-axis skew error
sl.ganvgn	Split-axis velocity feed-forward gain
sl.settim	Settling time in ticks; reads negative until motion completes
sl.maxerr	Maximum previous position error

Related Routines

sl.listdata
sl.getdata
sl.defwave
sl.wave.trig

Calling Sequence

```
CALL sl.deftrig (mot.num, mode, sync, modes[], error)
```

Function

Used to specify what "trigger" is to be used to start data collection.

Input Parameters

`mot.num` Motor number

`mode` (Optional) Trigger mode as follows:
0 = OFF: No data collected. If currently collecting, buffer will be stopped.
1 = Re-Trigger Mode: Trigger is reenabled after buffer fills.
2 = Single Sweep Mode : Trigger is NOT reenabled after buffer fills.

`sync` (Optional) Synchronization mode as follows:
0 = Trigger now (free running)
1 = Sync to rising edge of square wave. If square wave is not commanded, trigger never occurs.
2 = External trigger. See the routine **sl.exttrig( )** for details.

Output Parameters

`modes[ ]` If the input arguments are undefined, a real array containing the current value of the mode arguments above:
modes[0] = Current value of **mode** argument
modes[1] = Current value of **sync** argument

`error` Standard success/error code

Details

Typically, the trigger will be an internally generated square or sine wave excitation of the motor. This command will behave consistently no matter what motor it is sent to on a given servo board, since the trigger is the same for all axes on a given board. Note that old data in the buffer is not lost until the next trigger actually occurs.

Executing this routine with no inputs defined will simply return the current values of **mode** and **sync**.

Related Routines

sl.defdata
sl.defwave
sl.exttrig

Calling Sequence

```
CALL sl.defwave (mot.num, wave.type, amplt, period, error)
```

Function

This program specifies the motor motion to make during data collection.

Input Parameters

`mot.num`	Motor number
`wave.type`	Type of wave: 0 = Square wave 1 = Sine wave
`amplt`	Amplitude of the wave in encoder counts. Set to 0 to disable wave.
`period`	Period of wave in seconds. Set to 0 to disable wave.

Output Parameter

`error`	Standard success/error code

Details

Once a square or sine wave has been defined and triggered, the specified motor will make continuous reciprocal moves based on **amplt** and **period**. The motion can be halted by calling **sl.defwave( )** with either the amplitude or period parameter set to 0.

This routine, along with **sl.getdata( )** and **sl.defdata( )** allows you to see all the servo statistics that are generated when a robot motion is made. See the example program **ex.servo.data( ) - Joint Motion Data Monitoring Example** for details.

Related Routines

sl.listdata
sl.defdata
sl.getdata
sl.wave.trig

Calling Sequence

```
CALL sl.drive (mot.num, accel, spd, dist, abs, error)
```

Function

This program places the servo code in position mode and commands a relative or absolute motion. Trajectory generation will be performed by the servo code, so this command will execute over many servo cycles.

The routine **sl.break( )** can be called to determine when the motion is complete.

The routine **sl.stop( )** can be called to abort a motion started with this routine.

Acceleration and speed are included to control the motion.

Usage Considerations

The indicated motor must be in ASYNC mode.

Input Parameters

`mot.num`	Motor number
`accel`	Amount of time to reach maximum speed, in milliseconds
`spd`	Motor speed in encoder counts/millisecond
`dist`	The distance (or position to which) the motor is to move, in encoder counts
`abs`	If TRUE, drive to the actual position If FALSE, drive to a relative position

Output Parameter

`error`	Standard success/error code

Details

This routine performs an operation similar to the V$^+$ DRIVE program instruction. However, the programmer has the option to make the move to an absolute position or relative to the current position of the motor.

If the **spd** or **accel** arguments are set to zero, then the servo trajectory generator will apply the default speed stored in opcode **sl.trjdvm** and the default acceleration stored in opcode **sl.trjdta**.

Related Routines

sl.c.drive
sl.slew
sl.mslew
sl.stop
sl.c.stop

Calling Sequence

```
CALL sl.enable (mot.num, enable, finish, error)
```

Function

This program enables/disables ASYNC mode for the specified motor.

Input Parameters

`mot.num`	Motor number
`enable`	If TRUE, enable ASYNC mode for the specified motor If FALSE, disable ASYNC mode for the specified motor
`finish`	If TRUE, the program will wait until completion of the last motion instruction for the specified motor before DETACHing and ATTACHing the robot.

Output Parameter

`error`	Standard success/error code

Details

ASYNC mode must be enabled before any routine in the Advanced Servo Library that requires ASYNC mode can be run. ASYNC mode must be disabled for normal V+ motion operations to continue. ASYNC mode is enabled/disabled on a motor-by-motor basis.

While a motor or motors are in ASYNC mode, V+ will continue to plan a motion for the motor(s). However, motion control is left to the application program using the Advanced Servo Library routines. The remaining motors on the robot will continue to function normally. That is, the robot can be commanded to move using the standard V+ motion commands.

> **NOTE:** ASYNC mode can also be disabled by issuing the V+ instruction DISABLE POWER. Subsequent power-up of the robot will enable SYNC mode.

Related Routine

sl.c.enable

Calling Sequence

```
CALL sl.exttrig (enable, group, error)
```

Function

This routine causes an "external" trigger to occur on all axes.

Usage Considerations

The routines **sl.setpt.cfg( )**and **sl.setpt.fill( )** must have been called previously to configure and fill the setpoint buffer prior to executing it with this routine.

Input Parameters

`enable` Enable flag: TRUE to enable, FALSE to disable.

`group` (Optional) Trigger group number to control (the default is 0):
> 0 starts a setpoint buffer
= 0 triggers data collection
< 0 stops a setpoint buffer

Output Parameter

`error` Standard success/error code

Details

This command is global, in that all axes are affected identically, even across the servo task boundaries. External triggers that occur while data is being collected are ignored. This guarantees that the buffer will fill each time a trigger is accepted. If the buffer is NOT collecting data, a new trigger causes all data currently in the buffer to be lost.

This routine is used with the servo setpoint buffer and may be used with the routine **sl.getdata( )** for gathering servo data.

Related Routines

sl.setpt.cfg
sl.setpt.fill
sl.deftrig

Calling Sequence

```
CALL sl.free (mtr.mask, enable, error)
```

Function

Places the desired motor(s) into 'free' mode if 'free' mode is allowed for the specified motor(s).

Usage Considerationss

This routine will automatically place the required channel into and out of ASYNC mode.

The “FREE mode digital signal” defined in SPEC is supported by this routine.

If disabling FREE mode, the motor(s) will automatically be placed back into SYNC mode (position servo mode) and the robot will be DETACHed and reATTACHed to the calling task.

Input Parameters

`mtr.mask` Bitmask indicating which joint(s) to free up (one bit per motor. Motor #1 is the LSB):

Motor	Bit	Binary Bitmask	Hex Bitmask
1	1	^B1	^H1
2	2	^B10	^H2
3	3	^B100	^H3
4	4	^B1000	^H4
5	5	^B10000	^H5
6	6	^B100000	^H6
etc.			

`enable` If TRUE, enable FREE mode for specified channel
If FALSE, disable FREE mode for specified channel

Output Parameter

`error` Standard success/error code. If FREE mode is not allowed (it is disabled in SPEC) for a requested motor, error code –609

(`*Illegal joint number*`) is returned. An informational message is typed to the monitor if the global flag **sl.adv.diag** is nonzero.

Details

WARNING: Placing a gravity-loaded joint into FREE mode could cause the joint to drop, causing damage to the payload or the workspace.

FREE mode releases servo control of the motor by either disabling the Drive Enable signal for the specified channel, or by commanding zero torque. The specific method depends on the drive configuration. This mode is intended to allow the mechanism to be easily moved by hand during setup, or to create a compliant robot joint or joints to ease part insertion.

See the example routine **ex.free( ) - FREE Mode Example** for more details.

Related Routine

sl.c.free

Calling Sequence

```
CALL sl.getdac (mot.num, select, dac, error)
```

Function

Returns the current DAC value being output to the selected motor.

Input Parameters

`mot.num`	Motor number
`select`	If TRUE, **dac** is returned as a voltage (±10) If FALSE, **dac** is returned in DAC units (±32,767)

Output Parameters

`dac`	Current DAC output (in DAC counts or volts depending on the value of **select**)
`error`	Standard success/error code

Details

This function returns the current value of the DAC output to the robot amplifiers. Since there normally is some steady-state "holding" torque applied when joint is servoing in place, this value will generally be 0 only when arm power is disabled or the motor is uncalibrated. (In the case of velocity-style drives, 0 indicates no commanded velocity.)

Related Routines

sl.c.getdac
sl.setdac
sl.c.setdac

Calling Sequence

```
CALL sl.getdata (mot.num, buf.wait, opcode, first.rec,
                                    numrec, out[], error)
```

Function

This routine returns servo data for variables defined in the routine **sl.defdata( )**. It can also retrieve data from the force buffer.

Usage Considerations

The number of records returned may be restricted by the memory on the motion board. The error code –730 (`*Information not available*`) will be returned if data is requested before the trigger has been set for data collection.

Input Parameters

`mot.num` Motor number. Set to zero (0) to read force data.

`buf.wait` 0 = Stop buffer immediately and return data
1 = No-wait mode. Return –668 (`*Device in use*`) error if buffer not full
2 = Wait until buffer is full before reading the records

`opcode` Opcode variable of the data item to retrieve. The array **user.ind[ ]** defined in the call to **sl.defdata( )** contains the symbolic names of the data items that are to be collected. This value specifies which data item to return based on the definition of that array.

`first.rec` First record number to be read

`numrec` Number of records to be read

Output Parameters

`out[ ]` Array of the records that were read from the motion board, starting at element 0

`error` Standard V+error code. If >0, contains the number of elements in the **out[ ]** array.

Details

This routine, along with **sl.defdata( )** and **sl.listdata( )**, allows you see all the servo or force statistics that are generated when a robot motion is made. The routine **sl.defdata( )** must have been called previously to initiate data collection.

See the example program **ex.servo.data( ) - Joint Motion Data Monitoring Example** for details.

Related Routines

sl.listdata
sl.defdata
sl.defwave
sl.deftrig

Calling Sequence

```
CALL sl.hand.ctrl (mask, state, enable, error)
```

Function

Operates the hand solenoids.

Input Parameters

`mask`	A bitfield describing which solenoids to affect. With this input, it is possible to limit operation to a subset of solenoid lines. The LSB (Least Significant Bit) affects "Hand Solenoid #1". The next LSB affects "Hand Solenoid #2", etc. Solenoids with no bits set in the mask will not change state.
`state`	A bitfield that sets the state of each solenoid. (1 = energize, 0 = de-energize)
`enable`	(Optional) Boolean that enables the solenoids. TRUE to enable solenoids; FALSE to disable solenoids. This argument is ignored if the SELECTed robot does not support this feature. See the Details section below.

Output Parameter

`error`	Standard success/error code

Details

Some robots have a "Solenoid Enable" bit that turns on air pressure to the valve block. This bit is controlled by the **enable** argument. Robots that don't have this feature ignore the **enable** argument.

Different types of robots have different numbers of hand solenoids. The LSB of **mask** and **state** always correspond to the first solenoid. Higher order bits for which there are no corresponding available solenoids are ignored.

Calling Sequence

```
CALL sl.here (mot.num, position, error)
```

Function

This program sets the encoder register (position reference) to the indicated value and marks the motor as "calibrated" without moving the motor.

Usage Considerations

This routine requires ASYNC mode.

Input Parameters

`mot.num`	Motor number indicating which register to set
`position`	Encoder value to place in the register

Output Parameter

`error`	Standard success/error code

Details

Without moving the motor, the encoder value is set to the indicated value. This is useful, for example, in custom calibration programs that require reading a resolver or absolute encoder to get the position without moving the motor. After computing the correct position in encoder counts, this routine can be called to set the position reference.

WARNING: Any existing calibration will be invalidated, but the robot will still be considered to be calibrated and able to move.

Related Routines

sl.c.here
sl.adjust
sl.c.adjust
sl.where
sl.c.where

Calling Sequence

```
CALL sl.init ($password[], status[n])
```

Function

This program defines the global data structures used by the Advanced Servo Library.

Input Parameter

`$password[ ]` (Optional) Array of passwords, indexed by robot number, corresponding to the passwords assigned in each robot's SPEC parameter file. **$password[1]** corresponds to the first robot, **$password[2]** corresponds to the second robot, etc. If a password is incorrect or not specified for a robot protected by a password, the Advanced Servo Library routines are restricted to **read-only** access and an informational message is output to the Monitor window. The password checking is NOT case sensitive.

If password protection is not in effect for a robot, the corresponding array element is ignored and can be left undefined.

`status[n]` **[REQUIRED]** Real array, indexed by the robot number (1, 2, 3 ...) indicating initialization success or failure for the respective robot as follows:

status[n] = 0 READ ONLY access to robot #n.

status[n] = –1 Successful initializaton of robot #n. Full read/write access granted.

status[n] = –2 Initialization FAILED for robot #n.

Details

This program must be called before any of the routines in the Advanced Servo Library are called. This routines does not enable ASYNC mode for any motors.

Error messages are output to the monitor and execution of the subroutine is stopped if any of the following conditions exist:

- This edition of the ASL is incompatible with the current version of V^+.
- The SPECSUB.LIB library is not already loaded in memory, or its disk file cannot be found to lad.
- The current edition of SPECSUB is not compatible with the current V^+ system.

- The Adept MV controller is not configured to run a robot.
- The V+ system does not have an Enhanced Trajectory Control license installed.

Related Routine

sl.asl.cleanup

Calling Sequence

```
CALL sl.jnt.to.mtr (jnt.angles[], mtr.cnts[])
```

Function

This routine converts joint units (degrees or millimeters) into motor counts.

Input Parameter

`jnt.angles[ ]` Array of joint angles (beginning at index 0)

Output Parameter

`mtr.cnts[ ]` Array of corresponding motor counts (beginning at index 0)

Details

This routine accepts an array of joint values (in degrees or millimeters) as returned from the V+ SOLVE.ANGLES instruction and returns the corresponding motor count values.

Related Routine

sl.mtr.to.jnt

Calling Sequence

```
CALL sl.listdata (mot.num, out[], error)
```

Function

This program returns the status of the data collection being done by the motion board.

Input Parameter

`mot.num` Motor number

Output Parameters

`out[ ]` Array of output data as follows:
[0] = Returns length of data collection, in milliseconds
[1] = Milliseconds between updates of data
[2] = Buffer status:
0 = Data collection is disabled
1 = Waiting for trigger
2 = Collecting
3 = Full, halted (single sweep mode)
[3] = User's index for first data collection variable

.
.
[n] = User's index for last data variables

`error` Standard success/error code. If success, contains the number of elements in the **out[ ]** array.

Details

This routine, along with **sl.defdata( )** and **sl.getdata( ),** allows you to see all the servo statistics that are generated when a robot motion is made. See the example program **ex.servo.data( ) - Joint Motion Data Monitoring Example** for details.

Related Routines

sl.getdata
sl.defdata
sl.defwave
sl.wave.trig

Calling Sequence

```
CALL sl.mslew (board, mtr.data[], num.mtrs, abs, enc.pos[],
                                              error)
```

Function

This routine allows you to drive a series of joints (from 1 to all the joints in a servo task). It also allows you to read the current values of the encoder registers for all the joints on the specified board within the same servo tick.

Usage Considerations

The referenced motors must be in ASYNC mode.

Input Parameters

`board` First board in a servo task. For example, if boards 1 and 2 are assigned to CPU 1 and boards 3 and 4 are assigned to CPU 2, this routine can move all the motors on either boards 1 and 2 or boards 3 and 4. You cannot move motors on boards 1 and 3, or 2 and 4, nor can you move motors based on robot assignment.

`mtr.data[ ]` Array of servo database numbers and position values:
[1] = servo DB number
[2] = position value
[3] = servo DB number
[4] = position value
.
.

This array must start at index 1. See the Details section below for a description of the servo database allocation.

`num.mtrs` Number of motors to move (allows you to input an array with more elements defined than you want to use). If this value is 0, no motion will be occur, but encoder values will be returned.

`abs` If TRUE, position data is absolute; if FALSE, position data is relative

Output Parameters

`enc.pos[ ]` Array of encoder values (starts at index 0, one element per encoder)

`error` Standard success/error code. If successful, contains the number of elements in the **enc.pos[ ]** array.

Details

This routine allows you to move multiple axes to absolute or relative encoder positions. The axes will make a coordinated move. This routine steps axes in a single servo task, NOT joints on a single robot. This routine can be made to affect all the joints of a single robot only if the joints on that robot happen to be mapped to the same servo task.

> **NOTE:** The motion will be made across one servo cycle without any accel/decel profiling. Be careful to not pass values that would result in large motions.

The maximum allowable range of position values is:

-2^{24} to 2^{24-1}

The number of axes that step to new positions is controlled by the number of inputs provided. The inputs must be provided in pairs such that the first number is a servo database number and the next setpoint. No more than "n" pairs (2*n array elements) can be sent, where "n" is the number of servoable axes/board. If no setpoints are supplied, then no axis motion will be performed, but the encoder readings will still be provided. The input setpoints and output encoder values are all guaranteed to be applied/read from the same servo tick (1 or 2 milliseconds)

Servo databases are allocated starting with the first channel on the first servo board assigned to the servo task (there is one servo task per CPU). Motion board-to-CPU assignment is made using the CONFIG_C.V2 utility program. Servo databases can therefore be determined from hardware channel numbers. For example, in a two MI-6 system with the MI-6s addressed as servo board 1 and 2, and with both mapped to the servo task on the main CPU board, the servo database allocation would be as follows:

Servo Board	Board Channel	CPU Database
1	1	1
1	2	2
1	3	3
1	4	4
1	5	5
1	6	6

Servo Board	Board Channel	CPU Database
2	1	7
2	2	8
2	3	9
2	4	10
2	5	11
2	6	12

Related Routines

sl.slew
sl.drive
sl.c.drive
sl.rw.jmap

Calling Sequence

```
CALL sl.mtr.to.jnt (mtr.cnts[], jnt.angles[])
```

Function

This routine converts motor counts into joint units (degrees or millimeters).

Input Parameter

`mtr.cnts[ ]` Array of motor counts, starting at element 0

Output Parameter

`jnt.angles[ ]` Array of corresponding joint angles, starting at element 0

Details

This routine accepts an array of motor-count values and returns the corresponding joint values (in millimeters or degrees as appropriate).

Related Routine

sl.jnt.to.mtr

Calling Sequence

```
CALL sl.over.travel (mot.num, pos, neg, error)
```

Function

This program reads the state of the positive and negative overtravel switches. (The state of **both** switches is available only on the VMI.)

Input Parameter

`mot.num`	Motor number

Output Parameters

`pos`	State of the positive overtravel switch
`neg`	State of the negative overtravel switch
`error`	Standard success/error code

Details

This routine reads the state of the overtravel switches. The output parameters **pos** and **neg** are set to TRUE if the respective switch is on and to FALSE if the respective switch is off.

> **NOTE:** The MI-6/MI-3 boards have only one overtravel input per axis. In this case, both **pos** and **neg** output arguments will reflect the state of the single overtravel input.

Calling Sequence

```
CALL sl.pcocfg (mot.num, pco.state, pco.pos, delta, equ,
                                        frm.servo[], error)
```

Function

This program configures the Position Compare feature in the servo code. Full support is available only on VMI-based motion systems. There is reduced functionality on the VJI belt encoder channels 7 and 8. It is NOT supported on MI-6 or MI-3 motion boards.

Input Parameters

`mot.num` Motor number

`pco.st` Initial PCO State (VMI ONLY)
< 0 - No change
= 0 - Clear PCO
> 0 - Set PCO

`pco.pos` Initial compare position, in encoder counts

`delta` New compare delta, in encoder counts

`equ` New EQUal signal configuration. Bit mask as follows ("xx" means "don't care"):

Binary Bitmask	Hex Bitmask	PCO Operation
00xx	^H0	Nothing. PCO controllable with **pco.st**. (VMI ONLY)
0100	^H4	PCO pulses when motor position matches the PCO position **pco.pos**
0101	^H5	PCO toggles when motor position matches the PCO position **pco.pos**
0110	^H6	PCO is deasserted when motor position matches the PCO position **pco.pos**
0111	^H7	PCO is asserted when motor position matches the PCO position **pco.pos**

Binary Bitmask	Hex Bitmask	PCO Operation
10xx	^H8	Fire Camera Strobe #0 (CSTRB0) when motor position matches the PCO position **pco.pos**
11xx	^HC	Fire Camera Strobe #1 (CSTRB1) when motor position matches the PCO position **pco.pos**

Output Parameters

`frm.servo[ ]` Real array defined as follows:
[0] = Previous state of PCO output for this motion board
= 0 PCO was cleared
> 0 PCO was set
[1] = Previous compare position for this axis
[2] = Previous compare delta, in encoder counts
[3] = Previous compare configuration for this axis

`error` Success/error code. If successful, contains number of elements in the **frm.servo[ ]** array.

Details

The Position Compare Output (PCO) function generates an internal signal called "Equal" (EQU) when the actual position of the axis exactly matches the target position. This signal may be routed either to the physical PCO (on the Machine connector on the VMI ONLY) or to one of the backplane Camera Strobe lines (CSTRB0 and CSTRB1). After a trigger is generated, the Position Compare function will automatically update the target position by adding the input parameter **delta** to it. (See Note #3 below.)

On the VMI, the input argument **pco.st** controls the initial state of the physical PCO signal. That is, when this routine is called, the PCO signal will be set to the state indicated by the **pco.st** argument. If the application requires that the PCO signal be asserted when the axis passes over the trigger position (**pco.pos**), make sure that the initial PCO state is cleared when this routine is called (**pco.st** = 0).

NOTES:

1. On the VMI, each channel produces an EQU signal, but there is only one physical PCO per board, and no matter how it is configured, the PCO will always be directly controlled with the **pco.st** argument. The PCO tracks the most recent change in state of any of its available trigger sources (position compare triggers or **pco.st**). If two sources change at exactly the same time, clearing the PCO signal has priority over asserting the PCO signal.
2. V^+ can use the PCO function to generate camera strobes. Changing this configuration on a belt encoder may therefore interfere with the operation of the camera strobes.
3. The compare delta is unidirectional. That is after the initial compare position causes a match (a PCO trigger), the input parameter **delta** is added to it to form a new compare position. The old compare position will not trigger any changes in the PCO or camera strobe lines should the axis back up over it again.

   To force the system to trigger using the old position, simply call this routine again to reconfigure the PCO before backing over the position.
4. The VJI supports the Position Compare function on its belt encoder channels only, and has no physical PCO output, so several of the options described above are not available on its belt channels. The functions labeled "VMI ONLY" do not apply to the VJI belt channels.

Related Routine

sl.c.pcocfg

Calling Sequence

```
CALL sl.read (mot.num, opcode, frm.servo, error)
```

Function

This program reads the value of the servo parameter indicated in the **opcode** argument.

Input Parameters

`mot.num` Motor number

`opcode` Symbolic name of parameter to read; see Details

Output Parameters

`frm.servo` Contains the value of the requested parameter

`error` Standard success/error code

Details

The following tables list the parameters that can be read. These parameter names should be entered as the **opcode** parameter. No other values should be entered for the **opcode** parameter.

Table 2-3. SPEC Servo Configuration Parameter Names

Parameter name (Opcode)	Data returned	Units
sl.dcgain	Proportional gain	None
sl.zero	Location of zero	None
sl.pole	Location of pole	None
sl.igain	Integrator gain	None
sl.emax	Maximum positional error to integrate	cts
sl.imax	Maximum value for integrator state	cts
sl.vfgain	Velocity feed-forward gain	None
sl.afgain	Acceleration feed-forward gain	None

Table 2-3. SPEC Servo Configuration Parameter Names (Continued)

Parameter name (Opcode)	Data returned	Units
sl.trqsgn	Motor sign	None
sl.maxtrq	Maximum DAC limit for motor	DAC cts
sl.mxdtrq	Duty Cycle Exceeded torque limit (RMS limit)	DAC cts
sl.dtylg2	Duty Cycle Exceeded filter pole	None
sl.sattim	Time before "*Motor stalled*" error declared	secs
sl.encsgn	Encoder sign	None
sl.enccfg	Encoder configuration byte	None
sl.inxcnt	Expected counts per zero index (0=Disable)	cts
sl.enverr	Soft envelope error tolerance	cts
sl.toltim	Milliseconds to be in tolerance before declaring it	msec
sl.dacfltr or sl.trqlg2	DAC Output Filter	None
sl.maxafg	Maximum acceleration feed-forward gain	None
sl.defafg	Default acceleration feed-forward gain	None
sl.velchk	Velocity envelope error limit	cts/msec
sl.menerr	Manual mode envelope error limit	cts
sl.setbcf	MI-6 axis-to-Board fault configuration	None
sl.henerr	Hard Envelope error tolerance	cts
sl.mantrq	Maximum torque to use in MCP manual mode	DAC cts
sl.minafg	Minimum acceleration feed-forward gain	None

Table 2-3. SPEC Servo Configuration Parameter Names (Continued)

Parameter name (Opcode)	Data returned	Units
sl.homcfg	Homing Configuration Byte. Least significant four bits interpreted as follows: 3210 ditt Bits 1 and 0 form a type field "tt" that is interpreted as: tt = 00 Calibrate using the home sensor tt = 01 Calibrate using the hard stop tt = 10 Calibrate to current position tt = 11 Absolute calibration (Reserved for Adept use only) Bit 2 forms an index field, indicating whether to calibrate to the nearest zero index or not: i = 1 Calibrate to home switch, hard stop, or current position i = 0 Calibrate to zero index Bit 3 forms a direction field, indicating which direction to search for the index: d = 1 Home to index moving ON sensor d = 0 Home to index moving OFF sensor	None
sl.sofint	Stop-on-force integral gain	None
sl.ganxgn	Gantry skew error gain	None
sl.ganvmx	Gantry maximum velocity feed-forward gain	None

Table 2-3. SPEC Servo Configuration Parameter Names (Continued)

Parameter name (Opcode)	Data returned	Units
sl.mchcfg	Machine input polarity. Bitfield as follows: Bit 0 Home sensor Bit 1 Positive Overtravel Bit 2 Negative Overtravel Bit 3 Drive Fault If bit is set (logical 1) then signal is configured as Active High; otherwise, signal is configured as Active Low.	None
sl.homin1	Maximum distance to search for first zero index	cts
sl.homin2	Calibration position at first zero index	cts
sl.homin3	Maximum width of home sensor, or hardstop tolerance, depending on calibration configuration.	cts
sl.homin4	Expected distance from home sensor to first zero index or absolute position of factory calibration (for Adept robots).	cts
sl.homin5	Park position after calibration is complete	cts
sl.homin6	Initial speed for home switch search	cts/msec
sl.homin7	Fine search speed for home switch	cts/msec
sl.homin8	*Motor Stalled* time-out for use during calibration	secs
sl.trjdvm	Default velocity for the DRIVE command.	cts/msec
sl.amptype	Amplifier type	None
sl.ctscom	Counts per commutation cycle. (VJI-controlled robots only)	cts

Table 2-3. SPEC Servo Configuration Parameter Names (Continued)

Parameter name (Opcode)	Data returned	Units
sl.comlg	Commutation lead gain (VJI-controlled robots only)	None
sl.scribe	Scribe position (AdeptOne family of robots only)	cts
sl.dcgain	DC gain value	None
sl.szero	S-Plane zero	None
sl.spole	S-Plane pole	None
sl.twacal	Commutation position at first index. (VJI-controlled robots only)	cts
sl.igain	Raw, unscaled integral gain	None
sl.latch	"Latched" errors word	None
sl.calgrp	Calibration group bit mask	None
sl.trjdta	Servo internal trajectory generator acceleration time in milliseconds. (Default = 500 milliseconds)	msec
sl.tollg2	Tolerance extrapolation time	secs

Table 2-4. SPEC Servo Data Parameter Names

Parameter name	Data Returned	Units
sl.caltrq	Actual (motor) torque command	DAC cts
sl.cmdvel	Commanded velocity	cts/msec
sl.velerr	Velocity error	cts/msec
sl.motvel	Actual (motor) velocity	cts/msec
sl.cmdpos	Commanded position (setpoint)	cts
sl.poserr	Position error	cts
sl.motpos	Actual (motor) position	cts
sl.eplpos	Last latched position of encoder	cts
sl.inxpos	Last position of zero index	cts
sl.inxdel	Actual count between zero indexes	cts
sl.inxerr	Running count of index errors	None
sl.inttrq	Integrator torque	DAC cts
sl.ganskw	Gantry skew error	cts
sl.ganvgn	Gantry velocity feed-forward gain	None
sl.settim	Settling time in ticks; reads negative until motion completes	None
sl.maxerr	Maximum previous position error	cts
sl.forcex	Force in X-direction	lbs
sl.forcey	Force in Y-direction	lbs
sl.forcez	Force in Z-direction	lbs
sl.torqx	Torque about X-axis	in-lbs
sl.torqy	Torque about Y-axis	in-lbs
sl.torqz	Torque about Z-axis	in-lbs
sl.intsta	Integrator state in torque counts	DAC cts

Table 2-4. SPEC Servo Data Parameter Names (Continued)

Parameter name	Data Returned	Units
sl.cmdacc	Commanded acceleration	cts/msec^2
sl.esttrq	Filtered torque	None
sl.sofset	Stop-on-force setpoint	cts
sl.eplflg	Latched Stop-on-Force flags	None
sl.sync	Axis SYNC/ASYNC/Command	None

Related Routines

sl.rw.*
sl.c.read
sl.c.write
sl.clr.ltch.err
sl.write

Calling Sequence

```
CALL sl.read.file ($fname, interactive, first.joint,
                                    num.joints, error)
```

Function

Reads an ASCII or Binary SPEC data file from the specified disk into RAM, overwriting the data for the currently SELECTed robot.

Usage Considerations

The servos will be restarted after the SPEC file is successfully read into RAM.

Current robot or joint specification data for the currently SELECTed robot will be lost at the succesful completion of this routine.

Input Parameter

`$fname` SPEC file name to read. May include an absolute or relative path. NFS shared disks are also supported. If the file extension is omitted, `.SPC` will be assumed.

`interactive` (Optional) If TRUE, this routine will describe the function of the read operation and prompt the operator for an input file name. The interactive text will be output to the Monitor window. If FALSE, the file name must be specified and the routine will not be interactive. Default is FALSE.

`first.joint` (Optional) If only joint data is to be read, this is the number of the first robot joint to be overwritten by the data in the file. If a standard robot file (`*.SPC`) is to be read, this value MUST be zero or undefined. As a special case, if this parameter is –1 , this routine will prompt the user for the number of the robot joint to overwrite, *regardless of the value of the interactive input parameter.* Default is 0.

`num.joints` (Optional) If only joint data is to be read, this specifies the number of joints to be read. If a standard robot file (`*.SPC`) is to be read, this parameter should be zero or undefined. Default is 0.

Output Parameter

`error` Indicates success or failure status as follows:
–1 = Operation failed early for some reason
+1 = Operation successful
V^+ Error code = hard error occurred

Details

WARNING: The specification data for the currently SELECTed robot will be lost at the successful completion of this routine.

This routine normally reads a data file that contains all of the accessible data for a given robot. Alternatively, it can be used to read a file that contains only the data for a specific joint or a series of sequential joints (a file with the `*.SPJ` extension contains joint data. In the latter mode, the data in the file will overwrite the data for a specified joint (or series of joints) for the selected robot. During this read, any non-joint specific data encountered (Reading a `*.SPC` file in JOINT mode) will generate an error.

When reading in joint mode, only the following cases of single or sequential joints can be input from a single `*.SPJ` data file:

1. Any single joint, coupled or uncoupled.
2. Any sequence of uncoupled joints.
3. A single, complete set of coupled joints. For instance, if there is a 3x3 matrix for three coupled joints, all three joints must be specified together and no extra joints can be specified.

Related Routines

sl.wrt.dat.file
sl.wrt.sys.file

Calling Sequence

```
CALL sl.read.opt.bit (word, bit, opt.set, opt.word, cpu)
```

Function

This routine returns the status of the requested option bit and, optionally, the entire option word.

Input Parameters

`word` Option word to interrogate as follows:
0 = Robot Option Word #1
1 = Robot Option Word #2
2 = System Option Word #1
3 = System Option Word #2
4 = Controller Option Word
5 = Processor Option Word
6 = Vision Option Word
7 = Force Option Word

`bit` Option bit to check inside of requested option word. Meaning of bit depends on which option word is being checked:

word = 0: ROBOT OPTION WORD #1
Meaning depends on the robot module. (See robot module documentation for the description of the robot option bits for your robot module.)

word = 1: ROBOT OPTION WORD #2

Bit	Description
1	Robot has RSC card
2	Robot has extended length quill
3	Robot has cleanroom option
4	Robot has HyperDrive option
5	Robot has high torque option
6	Reserved for future use
7	Robot has EC Certification option

word = 2: SYSTEM OPTION WORD #1

Bit	Description
1	V+ Extensions license installed
2	External encoders are supported
3-7	Reserved for future use
8	ALTER instruction enabled

word = 3: SYSTEM OPTION WORD #2

Bit	Description
1-6	Reserved for future use
7	Guidance Vision enabled
8	Inspection Vision enabled
9	DDCMP option installed
10-11	Reserved for future use
12	AdeptNet installed

word = 4: CONTROLLER OPTION WORD
Currently not used. All bits return 0.

word = 5: PROCESSOR OPTION WORD

Bit	Description
1	Processor is running V^+
2	Processor is running Vision code
3	Processor is running Servo code

word = 6: VISION OPTION WORD

Bit	Description
1-4	CPU number for VIS interface module
5	Reserved for future use
6	EVI present
7-8	Reserved for future use
9-10	Reserved for Adept internal use (may be 1 or 0)

word = 7; FORCE OPTION WORD

Bit	Description
1	Stop-on-force enabled

`cpu` (Optional) CPU number to read option bits for Processor Option

Output Parameters

`opt.set` TRUE if option bit is set; otherwise, FALSE.

`opt.word` (Optional) Requested option word

Details

This routine can be used to determine what system options are available to an application program.

Calling Sequence

```
CALL sl.rw.accel (mot.num, write, accel, error)
```

Function

This program reads or writes a robot Cartesian acceleration value.

Input Parameters

`mot.num`	Motor number
`write`	If TRUE, write value; if FALSE, read value
`accel`	If **write** is TRUE, new value for acceleration

Output Parameters

`accel`	If **write** is FALSE, returns current value of acceleration
`error`	Standard success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routine

sl.rw.jaccel

Calling Sequence

```
CALL sl.rw.amask (write, mask, error)
```

Function

This program reads or writes the robot axis mask.

Input Parameters

`write`	If TRUE, write value; if FALSE, read value
`mask`	If **write** is TRUE, new value for axis mask

Output Parameters

`mask`	If **write** is FALSE, returns value of the axis mask
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The axis mask indicates which axes are available to V+. There is only one value per robot for this parameter. The robot associated with this read/write operation is the currently SELECTed robot.

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Calling Sequence

```
CALL sl.rw.bcklsh (mot.num, write, pos.bcklsh, neg.bcklsh,
                                   error)
```

Function

This program reads or writes the mechanism backlash values.

Usage Considerations

Backlash values are always in encoder counts.

An error is returned if the currently SELECTed robot does not support backlash compensation. In this case, no data is read or written.

Input Parameters

`mot.num` Motor number

`write` If TRUE, write value; if FALSE, read value

`pos.bcklsh` If **write** is TRUE, new value for positive backlash

`neg.bcklsh` If **write** is TRUE, new value for negative backlash

Output Parameters

`pos.bcklsh` If **write** is FALSE, return current positive backlash value

`neg.bcklsh` If **write** is FALSE, return current negative backlash value

`error` Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines). Returns error if selected robot does not support backlash compensation.

Details

The values for these motion parameters will be changed in local RAM only. The values are not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Calling Sequence

```
CALL sl.rw.cal.file (write, $cal.file, error)
```

Function

Reads or writes the robot calibration program file name.

Input Parameters

`write`	If TRUE, writes value; if FALSE, reads value
`$cal.file`	Calibration program file name

Output Parameter

`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

Robot calibration is controlled by a program located, by default, in the subdirectory \CALIB\ on the boot disk. SPEC allows a system customizer to specify a V^+ application program that will perform custom calibration operations. There are special file and program naming conventions that should be adhered to. For more information, please refer to the ***AdeptMotion VME Developer's Guide.***

The values for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

Calling Sequence

```
CALL sl.rw.calcfg (mot.num, write, cal.type, cal.method,
                                   cal.dir, error)
```

Function

Configures the calibration parameters for the indicated motor.

Input Parameters

`mot.num` Motor number

`write` TRUE to write new home configuration
FALSE to read current home configuration

`cal.type` Calibration type:
0 = Calibrate using home sensor
1 = Calibrate using hard stop
2 = Calibrate to current position
3 = Absolute calibration (special)

`cal.method` Calibration method:
0 = Calibrate to zero index
1 = Calibrate to home switch or hardstop

`cal.dir` Calibration direction:
0 = Calibrate to zero index moving OFF the sensor
1 = Calibrate to zero index moving ON the sensor

Output Parameters

`cal.type` Calibration type (see above)

`cal.method` Calibration method (see above)

`cal.dir` Calibration direction (see above)

`error` Standard success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The values for these motion parameters will be changed in local RAM only. The values are not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Calling Sequence

```
CALL sl.rw.collision (write, offsets[], first, last, error)
```

Function

This program reads or writes the robot collision avoidance offsets.

Usage Considerationss

1. This feature is supported ONLY for the XY3ZØ device module. (Device Module ID #21)
2. There must be **two** XY3ZØ robots running on the same controller for this function to have any effect. Robot option bit #5 must be set on both robots in order to enable collision detection.

Input Parameters

`write` TRUE to write values; FALSE to read values

`offsets[ ]` Real array containing six values corresponding to the six degrees-of-freedom describing the collision detection zone.

`first` (Optional) First element to read/write in the **offsets[ ]** array. Default is 0.

`last` (Optional) Last element to read/write in the **offsets[ ]** array. Default is 5.

Output Parameters

`offsets[ ]` If **write** is FALSE, returns the current collision avoidance offsets as follows:
offsets[0] = X offset in millimeters
offsets[1] = Y offset in millimeters (Only offset currently supported)
offsets[2] = Z offset in millimeters
offsets[3] = RX offset in degrees
offsets[4] = RY offset in degrees
offsets[5] = RZ offset in degrees

`error` Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines). If the SELECTed robot is not a XY3ZØ robot, error code –533 `*Invalid hardware configuration*` will be returned. If Robot Option Bit #5 is not enabled, error code –804 `*Option not installed*` will be returned.

Details

This program will check to ensure that the SELECTed robot is an XY3ZØ device, and that Option Bit #5 is enabled.

The XY3ZØ robot module has been enhanced to provide a simple form of collision detection and avoidance for the special case of two XY3ZØ robots mounted above the same work surface. For this configuration, it is assumed that the two robots share a common Y-Axis and that the Y axes have the same origin (i.e., the same physical zero position) and direction.

In order to allow a clearance zone to be taken into consideration when testing for a collision, a special array of six values has been allocated. Nominally, these six values correspond to the six degrees of freedom (X, Y, Z, y, p, r). However, for the XY3ZØ robot module, only the second value is used (Y component). The absolute value of the second parameter specifies a desired clearance zone to be maintained between the two robots along the Y axis. **This clearance must be set in both of the XY3ZØ robot modules**.

Calling Sequence

```
CALL sl.rw.colsn.det (write, obstacle, shape, parms[],
                                        error)
```

Function

This programs reads or writes the parameters that define the shape, location and size of a static Cartesian obstacle that is used for the generalized collision detection feature of the robot modules.

Input Parameters

`write` TRUE to write collision detection parameters.
FALSE to read collision detection parameters

`obstacle` Obstacle number:
–1 = PROTECTED Obstacle #1
–2 = PROTECTED Obstacle #2
0 = Unused (ignored)
1 = USER defined Obstacle #1
2 = USER defined Obstacle #2
3 = USER defined Obstacle #3
4 = USER defined Obstacle #4

`shape` Shape of obstacle as:
0 = No obstacle
1 = Box
2 = Cylinder
3 = Sphere

`parms[ ]` If **write** is TRUE, array of obstacle definition parameters as:
Location:
parms[1] = Centroid: X position (mm)
parms[2] = Centroid: Y position (mm)
parms[3] = Centroid: Z position (mm)
parms[4] = Rotation: YAW (deg)
parms[5] = Rotation: PITCH (deg)
parms[6] = Rotation: ROLL (deg)

In addition to LOCATION data, the following parameters are required to define the size of the obstacle.

For a BOX:
parms[7] = Size: X-LENGTH (mm)
parms[8] = Size: Y-LENGTH (mm)
parms[9] = Size: Z-LENGTH (mm)

For a CYLINDER:
parms[7] = Size: DIAMETER (mm)
parms[8] = Size: Z-HEIGHT (mm)

For a SPHERE:
parms[7] = Size: DIAMETER (mm)

Output Parameters

`shape` Shape of obstacle specified in **obstacle** argument:
0 = No obstacle
1 = Box
2 = Cylinder
3 = Sphere

`parms[ ]` If **write** is FALSE, array of obstacle definition parameters for obstacle specifed in **obstacle** argument:

If **shape** is 0, output values are invalid.

If **shape** is 1 (Box):
parms[1] = Centroid: X position (mm)
parms[2] = Centroid: Y position (mm)
parms[3] = Centroid: Z position (mm)
parms[4] = Rotation: YAW (deg)
parms[5] = Rotation: PITCH (deg)
parms[6] = Rotation: ROLL (deg)
parms[7] = Size: X-LENGTH (mm)
parms[8] = Size: Y-LENGTH (mm)
parms[9] = Size: Z-LENGTH (mm)

If **shape** is 2 (Cylinder):
parms[1] = Centroid: X position (mm)
parms[2] = Centroid: Y position (mm)
parms[3] = Centroid: Z position (mm)
parms[4] = Rotation: YAW (deg)
parms[5] = Rotation: PITCH (deg)
parms[6] = Rotation: ROLL (deg)
parms[7] = Size: DIAMETER (mm)
parms[8] = Size: Z-HEIGHT (mm)

If **shape** is 3 (Sphere):
parms[1] = Centroid: X position (mm)
parms[2] = Centroid: Y position (mm)
parms[3] = Centroid: Z position (mm)

parms[4] = Rotation: YAW (deg)
parms[5] = Rotation: PITCH (deg)
parms[6] = Rotation: ROLL (deg)
parms[7] = Size: DIAMETER (mm)

`error` Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

This routine allows the definition of Obstacles or regions in the workspace that the mechanism needs to avoid. After defining the regions, the robot will not be able to be driven into or through the regions with either the MCP (under manual control) or with straight-line moves (under program control). If the user or a program attempts to drive the robot into an obstacle the following error message will occur:

–901 `*Obstacle collision detected*`

When a collision is detected during manual control, the following error will be displayed on the MCP screen:

```
*OBSTACLE COLLISION*
```

The SOLVE.ANGLES instruction and the INRANGE function both test for collisions with static Cartesian obstacles. In addition to the other error bits that this instruction and function return, the following bit will also be turned on if a collision is detected:

Bit 13 (counting from zero) -> collision detected

For SOLVE.ANGLES, this bit is returned in the **error** variable. For INRANGE, this bit is returned as part of the value forth function.

The robot's tool tip and tool mounting flange are automatically tested by the operating system to ensure that they will not collide with the objects defined by this routine. This testing is performed under the following circumstances ONLY:

1. When the robot is moved in JOINT, WORLD, or TOOL mode via the MCP.
2. When the destination of the next motion is being planned
3. During a straight-line motion

The testing is NOT performed for ANY motions made in ASYNC mode since trajectory generation is bypassed for those types of motions. This testing is also NOT performed for joint interpolated motions.

The location of the obstacle is defined with respect to the base reference frame of the robot when its BASE transformation is NULL.

Three different obstacle shapes are supported:

1. Box
2. Cylinder
3. Sphere

Six parameters control the position and orientation of the obstacles:

1. Centroid: X position (mm)
2. Centroid: Y position (mm)
3. Centroid: Z position (mm)
4. Rotation: YAW (deg)
5. Rotation: PITCH (deg)
6. Rotation: ROLL (deg)

In addition, the following parameters are required to define the size of the different obstacle shapes:

For a Box:

- Size: X-LENGTH (mm)
- Size: Y-LENGTH (mm)
- Size: Z-LENGTH (mm)

For a Cylinder:

- Size: DIAMETER (mm)
- Size: Z-HEIGHT (mm)

For a Sphere:

- Size: DIAMETER (mm)

Calling Sequence

```
CALL sl.rw.comp (mot.num, write, last.val, comp[], error)
```

Function

This program reads or writes the linear compensation table for the SELECTed robot. This will allow user programs to change the linear compensation values used in V+.

Usage Considerations

This routine requires an enhanced robot module that supports linear compensation.

Input Parameters

`mot.num` Motor number

`write` If TRUE, write values; if FALSE, read values

`last.val` If **write** is TRUE, total number of correction values to apply
If **write** is FALSE, total number of correction values to read (clipped at 1,010)

`comp[ ]` If **write** is TRUE, Linear Compensation table in the following format:

comp[1]	Lowest position for comp values to be corrected
comp[2]	Highest position in mm for corrections
comp[3]	Change in joint position per correction value
comp[4-9]	Future use. Set equal to 0
comp[10]	First correction value at the lowest position. These correction values are equal to: actual_position – commanded_position
comp[11]	Second correction value
	.
	.
comp[1010]	Last possible correction value

Input Parameters

`comp[ ]` If **write** is FALSE, returns compensation data

`error` Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The values for these motion parameters will be changed in local RAM only. The values are not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Calling Sequence

```
CALL sl.rw.cspd (write, cspd, error)
```

Function

This program reads or writes the maximum Cartesian pendant translation speed.

Input Parameters

`write`	TRUE = write value; FALSE = read value
`cspd`	If **write** is TRUE, new value for manual Cartesian speed

Output Parameters

`cspd`	If **write** is FALSE, current value of manual Cartesian speed.
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Calling Sequence

```
CALL sl.rw.ctol (mot.num, write, ctol, error)
```

Function

This program reads or writes the COARSE tolerance.

Input Parameters

`mot.num`	Motor number
`write`	If TRUE, write value; if FALSE, read value
`ctol`	If **write** is TRUE, new value for coarse tolerance

Output Parameters

`ctol`	If **write** is FALSE, returns coarse tolerance
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routine

sl.rw.ftol

Calling Sequence

```
CALL sl.rw.enccfg (mot.num, write, enc.cfg, error)
```

Function

This program will read or write the encoder configuration byte for the specified motor.

Input Parameters

`mot.num` Motor number

`write` TRUE to write new configuration; FALSE to read current configuration

`enc.cfg` If **write** is TRUE, new encoder configuration byte

Output Parameters

`enc.cfg` If **write** is FALSE, current encoder configuration byte

`error` Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines). –658 `*Device hardware not present*` is returned if an attempt is made to set a Rev 0 board up for an 8MHz edge rate.

Details

The encoder configuration byte is a bit field broken down as follows:

Bit	Hex Bitmask	Description
1	^H1	Phase A HIGH/LOW (Encoder phasing configuration)
2	^H2	Phase B HIGH/LOW (Encoder phasing configuration)
3	^H4	Index Active HIGH/LOW (Encoder phasing configuration)
4	^H8	Zero Index Checking ON/OFF
5	^H10	Quadrature Error Checking (ON/OFF)
6	^H20	Broken Encoder Wire Detection (ON/OFF)

Bit	Hex Bitmask	Description
7	^H40	Extra filtering (2MHz Max Edge Rate)
8	^H80	No filtering (8MHz edge rate). Available on Revision 1 (Enhanced MI-3/6 boards) **only**. Board **must** be configured as Rev 1.

Calling Sequence

```
CALL sl.rw.ftol (mot.num, write, ftol, error)
```

Function

This program reads or writes the FINE tolerance.

Input Parameters

`mot.num`	Motor number
`write`	If TRUE, write value; if FALSE, read value
`ftol`	If **write** is TRUE, new value for fine tolerance

Output Parameters

`ftol`	If **write** is FALSE, return fine tolerance
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routine

sl.rw.ctol

Calling Sequence

```
CALL sl.rw.hand (write, open.sig, close.sig, error)
```

Function

This program reads or writes the hand control signals. (Used by OPEN, OPENI, CLOSE, CLOSEI, RELAX.)

Input Parameters

`write`	If TRUE, write value; if FALSE, read value
`open.sig`	If **write** is TRUE, new value for "Open hand" control signal (if only one valve is used to control the hand, this value should be set to zero)
`close.sig`	If **write** is TRUE, new value for "Close hand" control signal

Output Parameter

`open.sig`	If **write** is FALSE, returns "Open hand" signal
`close.sig`	If **write** is FALSE, returns "Close hand" signal
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

There is only one value per robot for this parameter. The robot associated with this read/write operation is the currently SELECTed robot.

The values for these motion parameters will be changed in local RAM only. The values are not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Calling Sequence

```
CALL sl.rw.hdw.latch (mot.num, write, latch, error)
```

Function

This program configures the axis-specific encoder latching. This feature is only available on the Enhanced MI-3/MI-6 boards (Rev 1 servo boards).

This routine can be used to configure the standard board-level position latching available on all standard AdeptMotion boards (VMI, MI-3/MI-6, VJI, EJI). See **Details** section below.

Input Parameters

`mot.num` Motor number

`write` TRUE to write encoder latching configuration to V^+. FALSE to read encoder latching configuration from V^+.

`latch` Two-bit bitfield indicating which position latch signal to use for this motor as:

Binary Value	Hex Bitmask	Signal
^B00	^H0	None
^B01	^H1	POS_LATCH 1
^B10	^H2	POS_LATCH 2
^B11	^H3	Illegal Value

These signals are defined in CONFIG_C under Digital Signals.

Output Parameters

`latch` Two-bit bitfield indicating which position latch signal is configured for this motor as:

Value	Signal
00	None
01	POS_LATCH 1
10	POS_LATCH 2

`error` Standard success/error code. If illegal latch configuration is specified, error code –402 `*Illegal value*` is returned. (See **"Errors" on page 20** for details on special error codes for the `rw` routines.)

Details

There are three possible cases for encoder latching on AdeptMotion boards:

1. Old Rev 0 (VMI) boards will work as they always did with the new servo code. If an application program attempts to use this routine to enable axis-specific latching, the new setting will be accepted. however, it will affect all axes. This is how it behaved before. You can determine if you have a Rev 0 or Rev 1 board by calling the Advanced Servo Library routine **sl.version( )**.
2. In order to support field service spares of Rev 0 boards with Rev 1 boards, configuration of the Enhanced MI-6/MI-3 board to be Rev 1 will be performed with a hardware jumper. From the servo's perspective, if a Rev 1 board is configured to be Rev 0, the axis-specific latching will not be useable, but the board-wide latching should work as it would on a true Rev 0 board.
3. Rev 1 boards configured as Rev 1. These boards will have board-wide latching disabled and will use only axis-specific latching. *At boot, they will attempt to simulate board level latching by configuring all the axes identically based upon the CONFIG_C settings for the board.* This configuration can then be overridden from this routine on an axis-by-axis basis.

Calling Sequence

```
CALL sl.rw.id.data (write, opt.word, model.num,
                                    serial.num,$name, error)
```

Function

This program reads or writes the robot option word, model number, serial number, and robot name.

Usage Considerations

Arm power is disabled and the servos are restarted if the robot option word is changed.

Read-only access to Adept robots is allowed.

Input Parameters

`write`	If TRUE, write value; if FALSE, read value
`opt.word`	(Optional) If **write** is TRUE, new value for robot option word
`model.num`	(Optional) If **write** is TRUE, new value for robot model number
`serial.num`	(Optional) If **write** is TRUE, new value for robot serial number
`$name`	(Optional) If **write** is TRUE, new value for the robot name

Output Parameters

`opt.word`	If **write** is FALSE, current value of robot option word
`model.num`	If **write** is FALSE, current value for robot model number
`serial.num`	If **write** is FALSE, current value for robot serial number
`$name`	If **write** is FALSE, current value for robot name
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The value for these motion parameters will be changed in local RAM only. The values are not written to the system disk. See the robot's device module documentation for information on robot option words.

Calling Sequence

```
CALL sl.rw.jaccel (mot.num, write, accel, error)
```

Function

This program reads or writes a robot joint acceleration value.

Input Parameters

`mot.num`	Motor number
`write`	If TRUE, write value; if FALSE, read value
`accel`	If **write** is TRUE, new value for joint acceleration

Output Parameters

`accel`	If **write** is FALSE, returns joint acceleration
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routine

sl.rw.accel

Calling Sequence

```
CALL sl.rw.jmap (write, rob.num, jmap[], error)
```

Function

This program reads or writes the joint/external encoder mapping for the specified robot number.

Usage Considerations

The robot will be DETACHed after reading OR writing the joint mapping.

The robot will be left in an uncalibrated state after writing the new mapping.

Arm power will be disabled and the servos will be restarted after writing the new mapping.

Writing a new joint mapping to an Adept robot is not allowed.

Input Parameters

`write` If TRUE, write the joint map in **jmap[ ]**; if FALSE, read the joint mapping for the indicated robot and return the values in **jmap[ ]**.

`rob.num` Robot number to read/write joint mapping. To read/write external encoder mapping, set this value to zero.

`jmap[ ]` If **write** is TRUE, new joint map for specified **rob.num** as follows:

jmap[1] Joint/Encoder 1 Board number
jmap[2] Joint/Encoder 1 Channel number
jmap[3] Joint/Encoder 2 Board number
jmap[4] Joint/Encoder 2 Channel number
jmap[5] Joint 3 Board number
jmap[6] Joint 3 Channel number
.
.
jmap[n] Joint (n+1)/2 Board number
jmap[n+1] Joint (n+1)/2 Channel number

Output Parameters

`jmap[ ]` If **write** is FALSE, contains returned joint map with the same format as the input array above.

`error` Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines). If an attempt is made to read/write

external encoder mapping on a system that does not support external encoders, the error –658 (`*Device harware not present*`) is returned. No data will be read or written.

Details

To ensure that the correct number of parameters are passed into this routine, you should perform a read function for the specified robot, make the desired changes, and write the modified **jmap[ ]** array back to the servos.

The values for these motion parameters will be changed in local RAM only. The values are not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routine

sl.mslew

Calling Sequence

```
CALL sl.rw.link (link.dim.num, write, l.dim, error)
```

Function

This program reads or writes the robot link dimensions.

Usage Considerations

Changing the robot link dimensions will dramatically affect how the robot performs, as well as robot calibration. Please refer to the robot's device module documentation on how to properly interpret the link dimensions.

Input Parameters

`link.dim.num` Link dimension number

`write` If TRUE, write value; if FALSE, read value

`l.dim` If **write** is TRUE, new value for the link dimension

Output Parameters

`l.dim` If **write** is FALSE, returns value of the link dimension

`error` Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routine

sl.rw.tooloff

Calling Sequence

```
CALL sl.rw.max.c.acc (write, max.accel, error)
```

Function

This program reads or writes the robot maximum Cartesian acceleration.

Input Parameters

`write`	If TRUE, write new value; if FALSE, read value
`max.accel`	If **write** is TRUE, new value for maximum Cartesian acceleration

Output Parameters

`max.accel`	It **write** is FALSE, returns maximum Cartesian acceleration
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

There is only one value per robot for this parameter. The robot associated with this read/write operation is the currently SELECTed robot.

The values for these motion parameters will be changed in local RAM only. The values are not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routine

sl.rw.max.j.acc

Calling Sequence

```
CALL sl.rw.max.j.acc (mot.num, write, max.accel, error)
```

Function

This program reads or writes the robot maximum joint acceleration.

Input Parameters

`mot.num`	Motor number
`write`	If TRUE, write new value; if FALSE, read value
`max.accel`	If **write** is TRUE, new value for maximum joint acceleration

Output Parameters

`max.accel`	It **write** is FALSE, returns maximum joint acceleration for specified joint
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The values for these motion parameters will be changed in local RAM only. The values are not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routine

sl.rw.max.c.acc

Calling Sequence

```
CALL sl.rw.maxcspd (write, max.spd, error)
```

Function

This program reads or writes the maximum robot speed during Cartesian motions.

Input Parameters

`write`	If TRUE, write value; if FALSE, read value
`max.spd`	If **write** is TRUE, new value for maximum speed

Output Parameters

`max.spd`	If **write** is FALSE, returns maximum speed
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routine

sl.rw.maxjspd

Calling Sequence

```
CALL sl.rw.maxjspd (mot.num, write, max.spd, error)
```

Function

This program reads or writes the maximum robot speed during joint-interpolated motions.

Input Parameters

`mot.num`	Motor number
`write`	If TRUE, write value; if FALSE, read value
`max.spd`	If **write** is TRUE, new value for maximum joint speed

Output Parameters

`max.spd`	If **write** is FALSE, returns maximum joint speed
`error`	Success/error code (See **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routine

sl.rw.maxcspd

Calling Sequence

```
CALL sl.rw.mspd (mot.num, write, mspd, error)
```

Function

This program reads or writes maximum joint speed under manual control with the MCP.

Input Parameters

`mot.num`	Motor number
`write`	If TRUE, write value; if FALSE, read value
`mspd`	If **write** is TRUE, new value for manual joint speed

Output Parameters

`mspd`	If **write** is FALSE, returns manual joint speed
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routine

sl.rw.mtic

Calling Sequence

```
CALL sl.rw.mtic (mot.num, write, mtic, error)
```

Function

This program reads or writes robot pendant tic increment (smallest positional increment when using the speed pot on the MCP).

Input Parameters

`mot.num`	Motor number
`write`	If TRUE, write value; if FALSE, read value
`mtic`	If **write** is TRUE, new value for pendant tic increment

Output Parameters

`mtic`	If **write** is FALSE, returns current value of pendant tic increment
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routine

sl.rw.mspd

Calling Sequence

```
CALL sl.rw.njts  (write, n.jts, stt)
```

Function

This program will read or write the number of joints for the currently selected robot.

Input Parameter

`write` TRUE = write value, FALSE = read value

`n.jts` If write, new value for number of robot joints

Output Parameter

`n.jts` If read, current number of robot joints

`stt` Success/failure status
0 = success
<0 = Failure (V^+ error code)

Details

If the number of joints is changed the servos will be restarted.

> **NOTE:** The maximum number of joints is limited to the max supported by the kinematic device module of the selected robot.

Calling Sequence

```
CALL sl.rw.pwr.decel (write, decel, error)
```

Function

This program reads or writes the robot "Arm Power Off Deceleration".

Usage Considerations

Robot Option Bit #12 must be set in Robot Option word #1.

Input Parameters

`write`	TRUE = write **decel** value FALSE = read **decel** value
`decel`	If **write** is TRUE, new value for deceleration

Output Parameters

`decel`	If **write** is FALSE, current value of deceleration
`error`	Standard success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Details

The "Arm Power Off Deceleration" is an additional parameter that is used when a trajectory is prematurely stopped and arm power is turned off. This value forces a specific deceleration percentage in the case where a severe error has occurred during a program-generated motion and arm power is disabled.

For most systems, this value is ignored since disabling arm power normally disables power to the amplifiers which makes trajectory generation meaningless. However, for systems that continue to supply power to the amplifiers for a period of time after arm power is disabled, this value can be used to specify a specific deceleration.

To have V+compute an appropriate deceleration, set this value to zero.

Calling Sequence

```
CALL sl.rw.rollover (mot.num, write, rollover, error)
```

Function

Reads or writes an encoder roll-over value.

Input Parameters

`mot.num`	Motor number
`write`	TRUE to write value; FALSE to read value
`rollover`	If **write** is TRUE, new value for encoder roll-over for this motor.

Output Parameters

`rollover`	If **write** is FALSE, current value for encoder roll-over for this motor.
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Details

The encoder roll-over value is typically used for continuous rotational joints, or for servo controlled unidirectional conveyors. Dynamically changing this value can prevent large motions in the event the encoder register rolls over.

Calling Sequence

```
CALL sl.rw.scale (write, enc.scales[,], error)
```

Function

This program reads or writes encoder scale factors for the currently selected robot.

Usage Considerations

The input array must be a GLOBAL DOUBLE or LOCAL DOUBLE array and must not contain any extra elements. See example below.

Some kinematic modules define more axes than you may actually have enabled. Default values will be returned for all possible axes, regardless of how many axes are actually enabled.

Input Parameters

`write` If TRUE, write values; if FALSE, read values

`enc.scales[,]` If **write** is TRUE, new values for encoder scale factors (must be a GLOBAL DOUBLE or LOCAL DOUBLE array)

Output Parameters

`enc.scales[,]` If **write** is FALSE, returns encoder scale factors (must be a GLOBAL DOUBLE or LOCAL DOUBLE array)

`error` Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

The values for these motion parameters will be changed in local RAM only. The values are not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

In order to maintain the encoder scale factor data structures, particularly for robots with coupling matrices, follow this procedure:

1. Call this routine with the **write** parameter set to FALSE. This will READ the current scale factor(s) for the selected robot.

2. Make the changes to the scale factors.

3. Call this routine again with the **write** parameter set TRUE, passing in the modified array.

This will ensure that the data received will be in the correct format for the robot.

This routine is limited to reading or writing 18 motors/joints.

> **NOTE:** This program reads/writes a two-dimensional array, where the left index is the joint number and the right indices point to the scale factors:

scale factor = enc.scales[jnt,idx]

Check the robot device module documentation for the definitions of the individual elements. For example, if joints 2 and 3 of a robot were part of a 2x2 coupling matrix, the two elements for joint 2 would be stored in:

`enc.scales[2,1]`= Motor 2 to motor 2 scale
`enc.scales[2,2]`= Motor 2 to motor 3 scale

and the elements for joint 3 would be stored in:

`enc.scales[3,1]` = Motor 3 to motor 2 scale
`enc.scales[3,2]` = Motor 3 to motor 3 scale

To get data for a single joint from the returned array, use the following code:

```
FOR ii = 1 TO LAST(enc.scales[mot.num,])
  jnt.scales[ii] = enc.scales[mot.num,ii]
END
```

The form of the array for a write operation must match exactly the form of the array returned from a read operation.

Example

```
; Change the first scale factor of joint 3 to 1123.64 and
; then display all the scale factors for joint 3

  LOCAL DOUBLE enc.scales[,]

; Make sure there are no defined elements for enc.scales[,]

  $task=$ENCODE(/I0,TASK())
  MCS "DELETER @"+$task+" enc.scales[,]"
```

```
; Read the current encoder scale factors

    CALL sl.rw.scales(FALSE, enc.scales[,], error)
    IF error < 0 GOTO 100

; Change the first scale factor for joint 3

    enc.scales[3,1] = 1123.64

; Write the new scale factors

    CALL sl.rw.scales(TRUE, enc.scales[,], error)
    IF error < 0 GOTO 100

; Display the scale factors for joint 3

    FOR ii = 1 TO LAST(enc.scales[3,])
      TYPE enc.scales[3,ii]
    END

100 RETURN
```

Calling Sequence

```
CALL sl.rw.softstops (mot.num, write, llim, ulim, error,
                                        lmin, umax)
```

Function

This program reads or writes both the working soft-stop limits and the maximum joint limits for the specified motor.

Input Parameters

`mot.num`	Motor number
`write`	If TRUE, write limits; if FALSE, read limits
`llim`	If **write** is TRUE, new Lower soft-stop limit in millimeters or degrees
`ulim`	If **write** is TRUE, new Upper soft-stop limit in millimeters or degrees
`lmin`	(Optional) If **write** is TRUE, new Minimum soft-stop limit in millimeters or degrees
`umax`	(Optional) If **write** is TRUE, new Maximum soft-stop limit in millimeters or degrees

Output Parameters

`llim`	If **write** is FALSE, returns lower limit for joint specified
`ulim`	If **write** is FALSE, returns upper limit for joint specified
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)
`lmin`	If **write** is FALSE, returns minimum limit for joint specified
`umax`	If **write** is FALSE, returns maximum limit for joint specified

Details

The values for these motion parameters will be changed in local RAM only. The values are not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Calling Sequence

```
CALL sl.rw.s_crv (write, prof.num, scrv[], error)
```

Function

This program reads or writes an S-curve profile. This allows dynamic redefinition of S-curve profiles.

Input Parameters

`write`	If TRUE, write the S-curve in **scrv[ ]**, else read the S-curve for the indicated joint and return the values in **scrv[ ]**.
`prof.num`	S-curve profile number to read/write (1 - 8). Clipped at 8.
`scrv[ ]`	If **write** is TRUE, new S-curve profile to write

Output Parameters

`scrv[ ]`	If **write** is FALSE, contains requested S-curve profile
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

There are five elements to an S-curve profile:

`scrv[1]` TRUE = enable profile, FALSE = disable profile
`scrv[2]` Time to ramp up to maximum acceleration
`scrv[3]` Time to ramp down to 0
`scrv[4]` Time to ramp up to maximum deceleration
`scrv[5]` Time to ramp down to 0

A maximum of 8 profiles can be defined per robot.

The values for these motion parameters will be changed in local RAM only. The values are not written to the system disk.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Calling Sequence

```
CALL sl.rw.tooloff (write, offset, error)
```

Function

This program reads or writes the robot tool offset.

Input Parameters

`write`	If TRUE, write value; if FALSE, read value
`offset`	If **write** is TRUE, new value for tool offset

Output Parameters

`offset`	If **write** is FALSE, returns value of the tool offset
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

There is only one value per robot for this parameter. The robot associated with this read/write operation is the currently SELECTed robot.

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk.

Related Routine

sl.rw.link

Calling Sequence

```
CALL sl.rw.tuning (mot.num, write, save, pgain, pzero,
                                   ppole, igain, ival,
                                   istep, affwd, vffwd,
                                   dacfltr,error)
```

Function

Reads or writes any or all the tuning parameters for the specified motor.

Usage Considerations

Motor tuning values may be changed. This could cause the motor to become unstable. USE EXTREME CAUTION.

Input Parameters

`mot.num`	Motor number
`write`	TRUE to write tuning parameters; FALSE to read tuning parameters
`save`	(Optional) If "TRUE", save the values when a new SPEC file is written or parameters are saved to the V+ Operating System. If FALSE, change only for current session. Default is FALSE.
`pgain`	(Optional) If **write** is TRUE, new value for Proportional Gain
`pzero`	(Optional) If **write** is TRUE, new value for Proportional Zero
`ppole`	(Optional) If **write** is TRUE, new value for Proportional Pole
`igain`	(Optional) If **write** is TRUE, new value for Integral Gain
`ival`	(Optional) If **write** is TRUE, new value for Maximum Integral Value
`istep`	(Optional) If **write** is TRUE, new value for Integral Step
`affwd`	(Optional) If **write** is TRUE, new value for Acceleration Feed Forward
`vffwd`	(Optional) If **write** is TRUE, new value for Velocity Feed Forward
`dacfltr`	(Optional) If **write** is TRUE, new value for DAC Output Filter

Output Parameters

`pgain`	(Optional) If **write** is FALSE, current value for Proportional Gain

`pzero`	(Optional) If **write** is FALSE, current value for Proportional Zero
`ppole`	(Optional) If **write** is FALSE, current value for Proportional Pole
`igain`	(Optional) If **write** is FALSE, current value for Integral Gain
`ival`	(Optional) If **write** is FALSE, current value for Maximum Integral Value
`istep`	(Optional) If **write** is FALSE, current value for Integral Step
`affwd`	(Optional) If **write** is FALSE, current value for Acceleration Feed Forward
`vffwd`	(Optional) If **write** is FALSE, current value for Velocity Feed Forward
`dacfltr`	(Optional) If **write** is FALSE, current value for DAC Output Filter
`error`	Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

CAUTION: Changing the motor tuning values can make the motor unstable. This could cause significant damage to the robot or the load.

Details

This routine allows access to any or all of the tuning parameters for the specified motor. If all the optional arguments are undefined, this routine does nothing and returns without error.

See the ***AdeptMotion VME Developer's Guide*** for detailed descriptions of all the motor tuning parameters.

Related Routines

sl.read
sl.c.read
sl.write
sl.c.write

Calling Sequence

```
CALL sl.set.gan.splt (mot.num, link, error)
```

Function

Configures the servos for a split axis joint. That is, creates a link between motor **mot.num** and the motor specified in the **link** argument.

Usage Considerations

This routine is for use with split-axis Gantry robot systems only.

Input Parameters

`mot.num` Motor number of the "master" motor

`link` Linkage pointer as follows:

0	Remove any existing linkage
–1	Put axis into slave mode
1-n	Link motor **mot.num** to motor "n"

Output Parameters

`link` Motor number of the linked channel BEFORE implementation of the new linkage. Returns 0 if there was no previous linkage.

`error` Success/error code (see **"Errors" on page 20** for details on special error codes for the `rw` routines)

Details

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

This routine sets/clears the pointer used by the skew-error compensator, and accepts the motor number of the other linked axis. Pointers for both channels are then set to point to each other.

Calling Sequence

```
CALL sl.setdac (mot.num, select, value, error)
```

Function

This program sets the indicated motor's DAC to the specified value.

Usage Considerations

The maximum output voltage is ±10 volts. The maximum DAC value is ±32,767.

The motor must be in ASYNC mode.

Input Parameters

`mot.num`	Motor number
`select`	(Optional) If TRUE, **value** contains a voltage (±10). If FALSE, **value** contains DAC counts (±32,767). Default is FALSE.
`value`	Desired analog voltage output, or DAC value to set. If undefined, the DAC is set to 0.

Output Parameter

`error`	Success/error code

Details

WARNING: This function could cause serious damage to the mechanism, tooling, or workspace or injury to personnel. Executing this function with arm power on may cause the axis specified to move at VERY HIGH SPEED. This command is open loop! **USE EXTREME CAUTION** when running this program.

Related Routine

sl.c.setdac

Calling Sequence

```
CALL sl.setpt.cfg (mtr, sz, group, wrap, open, abs, acc,
                                   dec, smooth.fact, error)
```

Function

This routine configures the position or torque setpoint buffer. This must be executed to size the buffer prior to calling the routines **sl.setpt.fill( )** or **sl.exttrig( )**.

Input Parameters

`mtr` Motor number for buffer configuration

`sz` Buffer size, in servo ticks
- < 0 Don't change
- >= 0 Buffer is cleared and resized to this value. Buffer size is limited by available V+ memory.

`group` Trigger group number to start/stop buffer
- < 0 Don't change
- 0 Disable triggering
- 1-12 Valid trigger number to use

Trigger groups can be set up to control multiple axes across the servo task boundaries. For example, configure axes 1-6 all the same using the same trigger number. When triggered, all axes will begin motion at the same time.

`wrap` Wrap flag
- < 0 Don't change
- 0 Single sweep. In this mode, the last value in the buffer will persist after completion.
- 1 Wrap around to beginning.

`open` Open loop flag
- < 0 Don't change
- 0 Closed loop (position setpoints)
- 1 Open loop (torque setpoints)

`abs` Absolute flag
- < 0 Don't change
- 0 Relative motions
- 1 Absolute motions

`acc` Ramp up time, in ticks. This applies only to "relative" buffers. The buffer will take this many ticks to reach its full value when started.

`dec` Ramp-down time, in ticks. This applies only to "relative" buffers. The buffer will take this number of ticks to ramp down after being stopped.

`smooth.fact` Velocity and acceleration smoothing factor. This value determines how much averaging is performed on the setpoints for the motion as follows:

<0 Don't change
0 Disable velocity smoothing computations
1 Compute average velocity based on 1 setpoint in the past and future.
2 Compute average velocity based on 2 setpoints in the past and future.
3 Compute average velocity based on 3 setpoints in the past and future.

etc

Output Parameter

`error` Standard success/error code

Details

This routine allows the user to write a position or torque setpoint stream directly to any motors in the system. When enabled, the setpoint buffer will command the servos at the servo rate (1kHz or 500Hz). This allows for tight path control for such tasks as electronic camming and scrubbing. Setpoint buffers for various motors can be triggered individually or simultaneously (including buffers running on different processors).

See the example program **ex.setpt( ) - Setpoint Buffer Example** for details.

Related Routines

sl.setpt.fill
sl.exttrig

Calling Sequence

```
CALL sl.setpt.fill (mtr, first, num, data[], error)
```

Function

This routine fills the setpoint buffer with either torque or position setpoints.

Usage Considerations

The routine **sl.setpt.cfg( )** MUST have been called to configure the buffer prior to execution of this routine. The routine **sl.exttrig( )** is used to actually run the setpoint buffer.

Input Parameters

`mtr`	Motor number
`first`	(Optional) first record to fill. The first record is #1.
`num`	(Optional) total number of records to fill. Default is 0.
`data[ ]`	Array of setpoints.

Output Parameter

`error`	Success/error code. If success, contains the number of elements in the buffer.

Details

See the example program **ex.setpt( ) - Setpoint Buffer Example** for details.

Related Routines

sl.exttrig
sl.setpt.cfg

Calling Sequence

```
CALL sl.slave.mode (axis.num, enable, error)
```

Function

This routine places the specified axis into or out of slave mode, such that one axis will follow the other's setpoints.

Usage Considerations

This routine is for use with split-axis Gantry robot systems ONLY.

ASYNC mode must be enabled for BOTH affected motors.

Input Parameters

`axis.num`	Axis number
`enable`	TRUE to enable slave mode; FALSE to disable slave mode

Output Parameter

`error`	Standard success/error code

Details

This routine can be used to allow coordination of split-axis pairs.

Related Routine

sl.set.gan.splt

Calling Sequence

```
CALL sl.slew (mot.num, enc.pos, error)
```

Function

This routine commands a step motion of a single motor.

Usage Considerations

The motion will be implemented over one V+ major cycle (16 milliseconds). The motor must be in ASYNC mode for this command to work.

> **NOTE:** The servos are expecting commands at the servo loop rate (either 1ms or 2ms, depending on configuration).

If the commands sent to this routine are not timed appropriately, the motion may be quite jerky.

Input Parameters

`mot.num`	Motor number
`enc.pos`	Absolute position (in encoder counts) to move to. The maximum range of position values is: -2^{24} to $+2^{24-1}$

Output Parameter

`error`	Success/error code

Details

CAUTION: The speed of the motor is directly proportional to the spacing of the setpoints. V+ monitor speed and program speed have no effect on the speed of a motor running in ASYNC mode. High acceleration rates can damage the motor windings, couplings, and other drive components.

This routine allows direct communication with the servo code to command motor motion. V+ normally checks the computed trajectory to ensure that the robot can reach the destination location safely. In ASYNC mode, these checks are bypassed. It is the responsibility of the programmer to ensure that the setpoints computed by a custom V+ trajectory algorithm will result in safe controlled motion of the motor.

The speed of the motor is directly proportional to the spacing of the setpoints. That is, the closer together the setpoints are, the slower the motor will move. The farther apart the setpoints are, the faster the motor will move. This is because the motion to the next encoder position occurs in one V^+ cycle.

Related Routines

sl.mslew
sl.drive
sl.c.drive

Calling Sequence

```
CALL sl.spare (spare.1, spare.2, error)
```

Function

Enables or disables the SPARE output signals on the motion board.

Usage Considerationss

If you do not wish to change the state of a SPARE signal, then leave it undefined.

Input Parameters

`spare.1` (Optional) If defined, the Spare signal #1 will be set to the appropriate state. TRUE to enable the signal, FALSE to disable the signal.

`spare.2` (Optional) If defined, the Spare signal #2 will be set to the appropriate state if the hardware supports it (MI-3/MI-6 ONLY). TRUE to enable the signal; FALSE to disable the signal.

Output Parameter

`error` Standard success/error code

Details

NOTE: The VMI has only one SPARE output signal. The MI-6/MI-3 has two SPARE output signals.

On the MI-6/MI-3, the SPARE_1 signal is located on the SERVO connector. The SPARE_2 signal is located on the MACHINE connector. See the ***AdeptMotion VME Developer's Guide*** for details on connector pinouts and signal specifications.

Calling Sequence

```
CALL sl.status (mot.num, j.status, error)
```

Function

This program reads the joint status bits for the specified motor.

Usage Considerations

The output parameter **j.status** must be declared as a DOUBLE or AUTO DOUBLE by the calling program.

Input Parameter

`mot.num` Motor number

Output Parameters

`j.status` Returns 32 bits of joint status information.

`error` Standard success/error code

Details

The status word bits are interpreted as follows (undefined bits are reserved and do not return useful information):

Bit	Hexadecimal Bitmask	Description
0	^H1	Home Switch state for this axis (input)
1	^H2	Spare Input state for this axis (input)
8	^H100	Spare Output state for this axis (output)
9	^H200	Brake release status for this axis (output)
10	^H400	Drive/motor enable status for this axis (output)
12	^H1000	High Power Enable state, all drives (output)
13	^H2000	Set if Hand solenoids are “active”
16	^H1 0000	“Multiple cycle” command complete (such as **sl.drive( )**)
17	^H2 0000	Set if motor is capable of servoing

24	^H100 0000	Motor is in tolerance
25	^H200 0000	First zero index seen after RESET or CAL
26	^H400 0000	Stop-on-force detected
30	^H4000 0000	Set if High Power is Enabled and Drive is Enabled
31	^H8000 0000	Motor has been calibrated

Related Routine

sl.c.status

Calling Sequence

```
CALL sl.stop (mot.num,error)
```

Function

Stop a motion started with a **sl.drive( )** or **sl.c.drive( )** routine call.

Usage Considerations

The indicated motor must be in ASYNC mode.

Input Parameter

`mot.num`	Motor number

Output Parameter

`error`	Standard success/error code

Details

This routine stops a joint using the normal deceleration phases of a motion (this is not an immediate halt of axis motion).

Related Routines

sl.c.drive
sl.c.stop
sl.drive

Calling Sequence

```
CALL sl.version (joint, $version[], version[], error)
```

Function

This program returns AdeptMotion VME version and revision information, as well as basic information on the selected joint.

Input Parameter

`joint` Joint number

Output Parameters

`$version [ ]` Array of strings as follows:
[0] = Servo Code version, revision, edit letter, and fork number
(Format: vvref Example: 122B2)

`version[ ]` Real array containing the following:
[0] Number of channels on board
[1] Axis type (1 = primary, 2 = secondary)
[2] Servo loop rate in Hz
[3] Max DAC for this motor
[4] Servo Board Type:
0 = VMI
1 = ACB
2 = VJI
3 = VFI
4 = MI-6 or MI-3
[5] Robot number that owns **joint**
[6] Motor number corresponding to **joint**
[7] Board number running **joint**
[8] Channel number running **joint**
[9] Processor # running servo code for **joint**
[10] Calibration compatibility code
[11] Robot serial number
[12] Servo mechanism ID
[13] Servo board revision

`error` Standard success/error code

Related Routine

sl.c.version

Calling Sequence

```
CALL sl.wave.trig (mot.num, sample.mode, sync, error)
```

Function

This program defines the trigger used for data collection. Typically, the trigger will be an internally generated square-wave excitation of the joint. Old data in the buffer is not lost until the next trigger actually occurs.

Input Parameters

`mot.num` Motor number

`sample.mode` Defines the sampling mode:
0 = Turn trigger OFF, no data collected
1 = Re-trigger, trigger is reenabled after buffer is filled
2 = Single sweep, trigger is not reenabled after buffer is filled

`sync` Defines the sync or start of the sampling
0 = Free run (start collection now)
1 = Sync to rising edge of square wave
2 = External trigger (see **sl.exttrig( )**)

Output Parameter

`error` Standard success/error code

Details

This routine, along with **sl.defdata( )** and **sl.getdata( )**, allows you to see all the servo statistics that are generated when a robot motion is made. See the example program **ex.servo.data( ) - Joint Motion Data Monitoring Example** for details.

Related Routines

sl.deftrig
sl.defwave
sl.exttrig
sl.getdata
sl.listdata

Calling Sequence

```
CALL sl.where (mot.num, cts, loc, error)
```

Function

Reads current location of motor **mot.num**.

Input Parameter

`mot.num`	Motor number

Output Parameters

`cts`	Current position in encoder counts
`loc`	(Optional) Current position in millimeters or degrees
`error`	Standard success/error code

Details

This routine returns the current position of the motor specified in the **mot.num** argument. It always returns the current position in encoder counts. If the **loc** argument is specified, the routine will also compute the current location in joint units (millimeters or degrees).

Related Routines

sl.c.where
sl.c.here
sl.here

Calling Sequence

```
CALL sl.write (mot.num, opcode, value, save, error)
```

Function

This program writes tuning and other SPEC parameters to memory.

Input Parameters

`mot.num`	Motor number
`opcode`	Symbolic name of desired value (see the description of **sl.read( )** for a list of the SPEC parameters that can be written)
`value`	New value for the specified parameter
`save`	If "TRUE", save the new value the next time a parameter file is written or parameters are written to the V+ operating system. If "FALSE", use this value only for the current session. (This routine will execute faster with this parameter set to FALSE.)

Output Parameter

`error`	Standard success/error code

Details

The value for this motion parameter will be changed in local RAM only. The value is not written to the system disk. If you wish to save the parameter to a disk file, or to the system disk using the routines **sl.wrt.dat.file( )** or **sl.wrt.sys.file( )**, respectively, the **save** parameter must be set to TRUE. Otherwise, the change is made only in RAM.

CAUTION: Be careful when writing motion parameters. They can have large effects on mechanism performance, longevity, and safety.

Related Routines

sl.rw.*
sl.read
sl.c.read
sl.c.write
sl.wrt.dat.file
sl.wrt.sys.file

Calling Sequence

```
CALL sl.wrt.dat.file (binary, interactive, $fname,
                                    first.joint, num.joints,
                                    error)
```

Function

This program writes all of the robot specification information, for the currently selected robot or specified joint, to a disk file in an ASCII or BINARY format.

Input Parameters

`binary` If TRUE, write a binary file; if FALSE, write an ASCII file

`interactive` If TRUE, this routine will be interactive, prompting the user to enter the disk file name. All text will be displayed in the Monitor window. If FALSE, this routine is noninteractive and all required input parameters must be specified.

`$fname` Name of file to create. Can include absolute or relative path. NFS shared disks are also supported. The file extension is optional. However, if it is undefined, it will be set to `.SPC`. If writing specification for a specific joint, or series of joints, the file extension will be set to `.SPJ` by default if undefined.

`first.joint` (Optional) If only joint data is to be written, this is the number of the first joint to be written. If all robot data is to be written, this value must be zero or undefined. As a special case, if this parameter is –1 this routine will prompt the user for the number of the joint, regardless of the value of the **interactive** parameter.

`num.joints` (Optional) If only joint data is to be written, this is the number of joints to write. If all robot data is to be written, this value must be zero or undefined.

Output Parameter

`error` Success/error code

Details

This routine normally writes all of the accessible data for a given robot. Alternatively, it can be used to write the data for a specific joint or series of joints. In this mode, general robot data (e.g., brake control I/O, number of joints, Cartesian motion times) is suppressed.

Although joint mode allows the data for multiple, sequential joints to be written to a single data file, WRITING THE DATA FOR MULTIPLE JOINTS INTO ONE DATA FILE SHOULD BE USED WITH EXTREME CARE. When writing a joint mode file, only the following cases of single or sequential joints should be written into a single data file:

1. Any single joint, coupled or uncoupled.
2. Any sequence of uncoupled joints.
3. A single, complete set of coupled joints. For instance, if there is a 3x3 matrix for three coupled joints, all three joints must be specified together and no extra joints can be specified.

Related Routines

sl.read.file
sl.wrt.sys.file

Calling Sequence

```
CALL sl.wrt.sys.file ($drive, interactive, error)
```

Function

This program saves all motion parameters for all robots to the system disk.

Usage Considerations

If this routine fails for any reason, the file block may be left in an unusable state.

Input Parameter

`$drive` Disk drive that contains the operating system file (the colon is optional).

`interactive` (Optional) If TRUE, this routine will interact with the operator exactly like SPEC. If FALSE, this routine is noninteractive. Default is FALSE. If TRUE, the **$drive** argument is ignored.

Output Parameter

`error` Standard success/error code.

Details

This routine writes the current motion parameters for all robots to the system disk.

If incorrect values for SPEC parameters are supplied, the write operation may fail.

WARNING: This routine, if executed in noninteractive mode (**interactive** = FALSE) will, without warning, write all currently defined motion parameters to the system disk. If the write operation fails for any reason, the system disk will be corrupted, forcing a reinstallation of the V+ Operating System. Before calling this routine, make sure you have a copy of the current SPEC data and system files.

Related Routines

sl.read.file
sl.wrt.dat.file

Advanced Servo Library Example Programs 3

ex.motion( ) - Robot Motion Example 164
ex.servo.data( ) - Joint Motion Data Monitoring Example 168
ex.setpt( ) - Setpoint Buffer Example 172
ex.follow( ) - Axis Following Examples 178
ex.j4.gainset( ) - Dynamic Tuning of Adept Joint 4 Example . . . 182
ex.calibrate( ) - Motor Calibration Example 185
ex.free( ) - FREE Mode Example 194

This chapter provides some example programs for using the Advanced Servo Library. The code for these files is in unprotected form in the file EXAMPLES.V2 on the Advanced Servo Library disk.

ex.motion( ) - Robot Motion Example

The following program shows the basic steps for initializing and moving a joint with the Advanced Servo Library.

```
.PROGRAM ex.motion(robot, num.mtrs, $password)

; ABSTRACT:  This example program shows how to initialize the Advanced
;       Servo Library and move a robot.  This routine performs the
;       following:
;
;               Enable ASYNCH mode
;               Drive the robot joints sequentially using sl.drive
;               Wait for joints to stop
;               Drive the joints simultaneously using sl.mslew
;               Disable ASYNCH mode
;               Delete the Advanced Servo Library data structures.
;
;       Note:  This code is written for a system with one robot and one
;       VMI board.  The robot uses the XYZ kinematic module.  Different
;       combinations of robots, VMIs, and kinematic modules will
;       require modification of this code to work correclty.
;
; INPUT PARMS:  robot        (Optional) Robot number to move.
;                               Default is Robot #1 if undefined.
;               num.mtrs     Number of motors to move on "robot"
;               $password    (Optional) Robot module password to
;                               access SPEC data.  May be undefined.
;
; OUTPUT PARMS: None
;
; SIDE EFFECTS: Robot motion will occur
;
; Copyright (c) 1995-97 by Adept Technology, Inc.

            AUTO abs, acc.t, board, dist, error, ii, spd.max
            AUTO REAL enc.pos[15], mtr.data[15], version[12]
            AUTO $a, $password[15], $version[2]
            LOCAL status[]

; Select the robot specified (robot must be calibrated and High
; Power must be enabled).
```

```
            IF NOT DEFINED(robot) THEN          ;If no robot was specified,
                robot = 1                       ; access the default robot
            END

;Do we need to select a new robot?

            IF SELECT(ROBOT) <> robot THEN
                DETACH ()
                SELECT ROBOT = robot            ;Select the robot
                ATTACH ()                       ;Attach the robot
            END

; Define the password(s) if required.

            IF DEFINED($password) THEN
                $password[robot] = $password    ;Use password passed in
            ELSE
                $password[robot] = ""           ;Assume no password
            END

; Call the Advanced Servo Library initialization routine.

            TYPE "Initializing the Advanced Servo Library..."
            CALL sl.init($password[], status[])
            IF NOT status[robot] THEN
                TYPE "Advanced Servo Library initialization failure ", /S
                TYPE "for robot #", /I0, robot, "."
                GOTO 140                        ;Exit
            END

; Enable ASYNCH mode for all joints.

            TYPE "Enabling ASYNC mode..."
            FOR ii = 1 TO num.mtrs
                CALL sl.enable(ii, TRUE, FALSE, error)
                IF error < 0 THEN
                        TYPE "Error for motor", ii, "  ", /S
                        CALL sl.adv.diag(error) ;Add error string
                        GOTO 120
                END
            END

            acc.t = 10              ;Time to reach max acceleration
            spd.max = 10            ;Maximum speed (encoder counts/ms)
            dist = 200              ;Distance to move (encoder counts)
            abs = FALSE             ;Make a relative move

; Drive each axis.

            TYPE "Driving each axis with 'sl.drive'..."
            FOR ii = 1 TO num.mtrs
                CALL sl.drive(ii, acc.t, spd.max, dist, abs, error)
                IF error < 0 THEN
                        TYPE "Error for motor", ii, "  ", /S
```

```
                        CALL sl.adv.diag(error)             ;Add error string
                        GOTO 120
                    END
                END

; Wait for the last axis to stop.

                TYPE "Waiting for all the motors to stop..."
                CALL sl.break(num.mtrs, error)
                IF error < 0 THEN
                    CALL sl.adv.diag(error)         ;Display error string
                    GOTO 120
                END

                TYPE
                PROMPT "Press enter to continue with the routine sl.mslew.  ", $a
                TYPE

; Generate motor data for "sl.mslew".

    50          mtr.data[1] = 1
                mtr.data[2] = -10
                mtr.data[3] = 2
                mtr.data[4] = -10
                mtr.data[5] = 3
                mtr.data[6] = -10
                mtr.data[7] = 4
                mtr.data[8] = -10

; Get the servo board number that the SELECT'ed robot is running on.
; This is needed for the subsequent CALL to sl.mslew.

                CALL sl.version(1, $version[], version[], error)
                IF error < 0 GOTO 140

                board = version[7]      ;Get board number from version array

; Move all motors simultaneously (motors must be in the same servo task).

                TYPE "Moving all motors with 'sl.mslew'..."
                FOR ii = 1 TO 1000
                    CALL sl.mslew(board, mtr.data[], num.mtrs, abs, enc.pos[],
error)
                    IF error < 0 THEN
                            TYPE "Error for motor", ii, "  ", /S;C03-
                            CALL sl.adv.diag(error) ;Add error string
                            GOTO 120
                    END
                END

; Disable ASYNCH mode for all axes.

   120          TYPE "Disabling ASYNC mode..."
                FOR ii = 1 TO num.mtrs
```

```
                CALL sl.enable(ii, FALSE, FALSE, error)
                IF error < 0 THEN
                        TYPE "Error for motor", ii, "  ", /S
                        CALL sl.adv.diag(error) ;Add error string
                END
            END

; Delete the Advanced Servo Library data structures.

   140      CALL sl.asl.cleanup()

            RETURN
.END
```

ex.servo.data( ) - Joint Motion Data Monitoring Example

This example program shows how to establish a data collection operation so you can monitor the behavior of an axis. This is similar to the operations that are performed with the SPEC utility program tuning options.

```
.PROGRAM ex.servo.data(robot, $password, mtr.num)

; ABSTRACT:  This program shows how to set up data collection.
;
; INPUT PARMS:  robot Robot number to test
;               $password (Optional) Robot module password to
;                                    access SPEC data.  May be undefined.
;               mtr.num Motor number to test on "robot"
;
; OUTPUT PARMS: None
;
; SIDE EFFECTS: None
;
; Copyright (c) 1995, 1996 by Adept Technology, Inc.

            AUTO buf.status, data.cycle, data[15], error, ii, num.data
            AUTO num.recs, out[128], smpmode, sync, test, update.rate
            AUTO wave.ampl, wave.period
            AUTO $password[15]
            LOCAL status[]

; Define the password(s) if required.

            IF DEFINED($password) THEN
                $password[robot] = $password    ;Use password passed in
            ELSE
                $password[robot] = ""           ;Assume no password
            END

; Call the Advanced Servo Library initialization routine.

            CALL sl.init($password[], status[])
            IF status[robot] <> -1 THEN
                TYPE "Advanced Servo Library initialization failure ", /S
                TYPE "for robot #", /I0, robot, "."
                GOTO 140                        ;Exit
            END

; Select a robot (robot must be calibrated and High Power must be enabled).

            DETACH (0)
            SELECT ROBOT = robot
            ATTACH (0)
```

```
; Enable ASYNC mode for specified joint.

            CALL sl.enable(mtr.num, TRUE, FALSE, error)
            IF error < 0 THEN
                TYPE "Error enabling ASYNC mode: ", error
                GOTO 120
            END

; Define test type.

            test = 0                ;Square wave test
            wave.ampl = 500         ;1,000 encoder count amplitude (+/- 500
counts)
            wave.period = 1.5       ;1.5 second wave period
            CALL sl.defwave(mtr.num, test, wave.ampl, wave.period, error)
            IF error < 0 THEN
                TYPE "Error enabling defining test: ", error
                GOTO 120
            END

; Define trigger mode and type.

            smpmode = 1             ;Use re-trigger mode
            sync = 1                ;Sync to rising edge of square wave
            CALL sl.wave.trig(mtr.num, smpmode, sync, error)
            IF error < 0 THEN
                TYPE "Error defining test trigger: ", error
                GOTO 120
            END

; Define data collection.

            data.cycle = 500        ;Collect for 500 ms
            update.rate = 1         ;Update every 1 ms
            data[1] = sl.poserr     ;Collect position error data
            num.data = 1            ;Number of data items to collect
            CALL sl.defdata(mtr.num, data.cycle, update.rate, num.data,
data[], error)
            IF error < 0 THEN
                TYPE "Error defining data collection: ", error
                GOTO 120
            END

; Show the actual status of data collection.

            CALL sl.listdata(mtr.num, status[], error)
            IF error < 0 THEN
                TYPE "Error verifying status of data collection: ", error
                GOTO 120
            END
            num.recs = status[0]/status[1]
            TYPE /C2, "Number of records being stored: ", num.recs
            TYPE "Data is updated every", status[1], " cycle(s)."
```

```
            TYPE "Buffer status code: ", status[2]
            TYPE "Data collection codes: ", /S
            FOR ii = 3 TO LAST(status[])
                TYPE status[ii], /S
            END
            TYPE

; Allow data collection for 5 seconds.

            WAIT.EVENT , 5

; Return the data.

            buf.status = 2          ;Wait until buffer is full
            CALL sl.getdata(mtr.num, buf.status, data[1], 1, num.recs, out[],
error)
            IF error < 0 THEN
                TYPE "Error returning data: ", error
                GOTO 120
            END

; Display the first 64 data points.  (Note: The number of points that
; can be returned with each call to "sl.getdata" depends on the
; memory available on the VMI.)

            FOR ii = 0 TO 63
                TYPE "Servo data ", /I2, ii, ": ", out[ii]
            END

; Turn off the wave test (set amplitude and period to 0).

            CALL sl.defwave(mtr.num, 0, 0, 0, error)
            IF error < 0 THEN
                TYPE "Error halting data collection: ", error
                GOTO 120
            END

; Wait for the last axis to stop.

            CALL sl.break(mtr.num, error)
            IF error < 0 THEN
                TYPE "Break: ", error
                GOTO 120
            END

; Disable ASYNC mode for all axes.

   120      CALL sl.enable(mtr.num, FALSE, FALSE, error)
            IF error < 0 THEN
                TYPE "Error disabling ASYNCH mode: ", error
            END

; Delete the Advanced Servo Library data structures.
```

```
   140      CALL sl.asl.cleanup()

            MCS "DELETER @ status[]"

            RETURN
.END
```

ex.setpt( ) - Setpoint Buffer Example

Using the setpoint buffer, you can predefine a series of robot points and then command the robot to move through these points.

```
.PROGRAM ex.setpt(robot, $password, circle)

; ABSTRACT:  This is an example routine that will move an XY pair of
;       axes along a circular or sine-wave path using the setpoint
;       buffer in the servos.
;
; INPUT PARMS:  robot (Optional) Number of robot to move.
;                       Default is robot #1.
;               $password  (Optional) Robot module password to
;                             access SPEC data.  May be undefined.
;               circle (Optional) If TRUE, follow circular path.
;                        Default is sine-wave path.
;
; OUTPUT PARMS: None
;
; SIDE EFFECTS: None
;
; Copyright (c) 1995-97 by Adept Technology, Inc.

            AUTO REAL circ, error, i, rad, size, stt
            AUTO loc
            AUTO $a
            LOCAL loc[], #ppoint[], $password[], status[]

; Select specified robot.

            DETACH ()   ;Robot must be detached before selecting new one

            IF NOT DEFINED(robot) THEN          ;If no robot was specified
                robot = 1                       ; access the default robot
            END
            SELECT ROBOT = robot    ;Select the robot

; Define the password(s) if required.

            IF DEFINED($password) THEN
                $password[robot] = $password    ;Use password passed in
            ELSE
                $password[robot] = ""           ;Assume no password
            END

; Initialize things.

            TYPE "Initializing the Advanced Servo Library..."
```

```
            CALL sl.init($password[], status[])
            IF status[robot] <> -1 THEN
                TYPE "Advanced Servo Library initialization failure ", /S
                TYPE "for robot #", /I0, robot, "."
                GOTO 100                        ;Exit
            END

            TYPE "Enabling ASYNC mode..."
            FOR i = 1 TO ID(3,10+robot)         ;ID() => number of motors
                CALL sl.enable(i, TRUE, FALSE, error)
            END

; Define the path locations.

            IF DEFINED(circle) THEN
                circ = circle       ;Used the passed value
            ELSE
                circ = FALSE        ;Default is sine-wave path
            END

            IF circ THEN            ;Compute setpoints for circle

                HERE loc            ;Record current location
                size = 1000         ;Buffer size = 1000 points
                rad = 20            ;Radius of circle
                FOR i = 0 TO size-1 ;Compute transforms for circle
                        SET loc[i] =
loc:RZ(360*i/(size-1)):TRANS(rad):RZ(-360*i/(size-1))
                END
                HERE #ppoint[0]     ;Record current joint locations

                ; Compute setpoints for servos and fill setpoint buffers.

                CALL ex.setpt.path(-1, size, #ppoint[], loc[], stt)

                TYPE "Triggering buffer..."
                CALL sl.exttrig(1)  ;Trigger buffers to start motion

            ELSE

                ; Configure setpoint buffers for individual sine waves
                ; on each motor.

                CALL ex.sin.wave(1, 1, 400, 0.51, 0.57, 0.57, stt)
                CALL ex.sin.wave(2, 0, 100, 0.7, 0.57, 0.57, stt)
                CALL ex.sin.wave(3, 1, 775, 0.3, 0.57, 0.57, stt)
                CALL ex.sin.wave(4, 0, 100, 0.4, 0.57, 0.57, stt)

                CALL sl.exttrig(1)  ;Trigger buffers to start motion

            END

            PROMPT "Press ENTER to abort motion.  ", $a
```

```
; Re-enabling SYNC mode will abort current motion.

            FOR i = 1 TO ID(3,10+robot)
                CALL sl.enable(i, FALSE, FALSE, error)
            END

   100      MCS "DELETEL @ loc[], #ppoint[]"
            MCS "DELETER @ status[]"
            MCS "DELETES @ $password[]"

            RETURN
.END


.PROGRAM ex.setpt.path(mask, num, #ppoint[], loc[], error)

; ABSTRACT:  Configures setpoint buffer for path following.  All points
;       in the path are stored as relative to the first point in the
;       path (either loc[0] or #ppoint[0], depending on whether loc[0]
;       is defined).
;
; INPUT PARMS:  mask         Bit mask indicating motors that should be
;                               configured to follow the path
;                               (-1 = all motors)
;               num          Number of points in the path
;               #ppoint[]    If following a path of precision points,
;                               locs[] must not be supplied, and this
;                               array defines the path with elements
;                               [0] to [num-1]
;               loc[]        If following a path of locations, only
;                               #ppoint[0] needs to be defined, and
;                               this array defines the path with
;                               elements [0] to loc[num-1]
;
; OUTPUT PARMS: error        Error code
;
; SIDE EFFECTS: None
;
; Copyright (c) 1995 by Adept Technology, Inc.

            AUTO flags, group, i, jt[10], locs, m, mots, mt[12], mts[12], out
            LOCAL mts[,]

            error = FALSE                       ;Assume success

            mots = ID(3,8)                      ;Number of motors in robot

            locs = DEFINED(loc[0])  ;Flag indicating defined locations path

            DECOMPOSE jt[] = #ppoint[0]         ;Stuff array of joint angles

            IF locs THEN            ;Solve for joint angles of ref loc
                SOLVE.ANGLES jt[], flags, error = loc[0], jt[], flags
            END
```

```
            CALL sl.jnt.to.mtr(jt[], mt[]);Joint angles => motor counts

            flags = SOLVE.FLAGS(jt[])          ;Get joint configuration flags

            FOR m = 1 TO mots       ;Compute motor group to move
                group = 0                      ;Disable unused motors
                IF (mask BAND BMASK(m)) OR (mask < 0) THEN
                        group = SELECT(ROBOT)
                END

                ; Configure the setpoint buffer.

                CALL sl.setpt.cfg(m, num, group, 1, 0, 0, 710, 710, 0, error)
                IF error < 0 GOTO 100
            END

; Fill setpoint buffers in blocks of 64 setpoints.

            out = 0
            WHILE out < num DO;Loop until all setpoints are done...
                i = 0
                WHILE (i < 63) AND ((out+i) < num) DO

                        ; Calculate mtr angles of desired positions.

                        IF locs THEN
                            SOLVE.ANGLES mts[], flags, error = loc[out+i],
jt[], flags
                        ELSE
                            DECOMPOSE mts[] = #ppoint[out+i]
                        END

                        ; Convert to motor angles.

                        CALL sl.jnt.to.mtr(mts[], mts[])

            ; Stuff motor angles into array correcting for absolute
            ; positions.

                        FOR m = 0 TO mots-1
                            mts[m,i] = mts[m]-mt[m]
                        END

                        i = i+1
                END

                ; Load the setpoint buffer.

                FOR m = 1 TO mots
                        IF (mask BAND BMASK(m+1)) OR (mask < 0) THEN
                            TYPE "Filling setpoint buffer for motor ", m
                            CALL sl.setpt.fill(m, out+1, i, mts[m-1,], error)
                            IF error == -402 THEN
```

```
                                PAUSE
                            END

                            IF (error < 0) AND (error <> -544) GOTO 100
                        END
                    END

                    out = out+i                     ;Count the completed block
                END

   100      MCS "DELETER @ mts[,]"

            RETURN
.END


.PROGRAM ex.sin.wave(mt, type, amp, freq, acc, dec, stt)

; ABSTRACT:  This routine configures and fills the setpoint buffer for
;       sine-wave or square-wave motion.
;
;  INPUT PARMS: mt      Motor number
;               type    Wave type as follows:
;                          0 = Closed-loop square wave
;                          1 = Close-loop sine wave
;                          2 = Open-loop sine wave
;               amp     Wave amplitude
;               freq    Wave frequency in Hz
;               acc     Ramp up time in seconds
;               dec     Ramp down time in seconds
;
; OUTPUT PARMS: stt     Error code
;
; SIDE EFFECTS: None
;
; Copyright (c) 1995 by Adept Technology, Inc.

            AUTO cycles, i, open, size
            LOCAL io[]

            open = ABS(type == 2)

            CASE type OF
              VALUE 0:                          ;Closed-loop sine wave
                size = STATE(14)/freq           ;STATE(14) returns servo loop
rate
                FOR i = 0 TO size-1
                        io[i] = -amp*SIGN(i >= size/2) ;Fill Buffer
                END

              ANY                                           ;Sine wave
                cycles = 1                      ;Number of cycles in buffer
                size = INT(STATE(14)/freq);Buffer size in ticks
                IF (size < 30) AND (INT(size) <> size) THEN;If high frequency
```

```
                        cycles = size
                        size = INT(STATE(14)*cycles/freq)
                    END

                    freq = STATE(14)*cycles/size ;Actual freq

                    FOR i = 0 TO size-1
                        io[i] = amp*SIN(360*cycles*i/size)
                    END
                END

                TYPE "Configuring buffer for motor", mt, "..."
                CALL sl.setpt.cfg(mt, size, SELECT(ROBOT), 1, open, 0,
acc*STATE(14), dec*STATE(14), 0, stt)
                IF stt < 0 GOTO 100

                TYPE "Filling buffer for motor", mt, "..."
                CALL sl.setpt.fill(mt, 1, size, io[], stt)
                IF stt < 0 GOTO 100

   100          MCS "DELETER @ io[]"

                RETURN
.END
```

ex.follow( ) - Axis Following Examples

This example program shows how to make one axis follow speed or position data from an external source. In this case the external source is another robot, but the source could be another encoder or data input from a serial or AdeptNet transmission.

```
.PROGRAM ex.follow()

; ABSTRACT:  This routine shows how to use the drive routine for a
;       "following" (electronic gearing) application.  This program
;       assumes that there are two robots in the system.  Robot 1 is
;       sent a normal V+ command to move one axes a random distance.
;       The DAC value for robot 1 is read and passed to a single axis
;       of the second robot, causing robot 2 to follow robot 1.
;
;       The first application is based on speed.
;
;           NOTE:  Unless carefully programmed, violent motions
;                  of the following robot may result.
;
;       The second application is based on motor position.
;
;           Each application will run until input signal 1001 is removed, or
;           low.
;
; INPUT PARMS:  None
;
; OUTPUT PARMS: None
;
; SIDE EFFECTS: None
;
; Copyright (c) 1995, 1997 by Adept Technology, Inc.

            AUTO cur.pos, dac, decel.rate, error, ii, ref.pos, scale, spd
            AUTO REAL enc.pos[15], mtr.data[15], start.pos, status[2]
            AUTO start

; Set the scale factor for the slave robot.

            scale = 1

; Call the Advanced Servo Library initialization routine.

            CALL sl.init(, status[])
            IF (NOT status[1]) OR (NOT status[2]) THEN
                TYPE "Error:  WRITE access to one of the robots is denied."
                PAUSE
            END

; Select Robot #1
```

```
            DETACH (0)
            SELECT ROBOT = 1
            ATTACH (0)

; Enable ASYNCH mode for joint 2 of robot 2.

            CALL sl.c.enable(2, 2, TRUE, FALSE, error)
            IF error < 0 GOTO 120

; Define a starting point for the first robot.

            SET start = HERE

; Remember the current postion of the second robot.

            CALL sl.c.where(2, 2, start.pos, , error);C05
            IF error < 0 GOTO 120

; ===================================================================
;
; This is a motor-following example based on speed.
; Robot 2 will track robot 1 until Signal 1001 is asserted.

            DO

                ; Move joint 1 of robot 1 (joint is assumed to be a
                ; translational joint).

                MOVE start:TRANS(,(RANDOM*10))
                DO

                        ; Get the DAC value for the first robot.

                        CALL sl.c.getdac(1, 2, FALSE, dac, error)
                        IF error < 0 GOTO 120

                        ; Drive the second robot.

                        CALL sl.c.setdac(2, 2, FALSE, dac*scale, error)
                        IF error < 0 GOTO 120
                UNTIL STATE(2) == 2
            UNTIL NOT SIG(1001)

            ; Decelerate the second robot.

            IF dac > 0 THEN
                decel.rate = -100
            ELSE
                decel.rate = 100
            END

            FOR ii = dac*scale TO 0 STEP decel.rate
                CALL sl.c.setdac(2, 2, FALSE, ii, error)
```

```
            IF error < 0 GOTO 120
        END

        ; Wait for the axis to stop by reading the motor velocity until
        ; it hits zero.

        DO
            CALL sl.c.read(2, 2, sl.motvel, spd, error)
            IF error < 0 GOTO 120
        UNTIL spd == 0

        ; Return second robot to original position.

        CALL sl.c.drive(2, 2, 1000, 50, start.pos, TRUE, error)
        IF error < 0 GOTO 120
        CALL sl.c.break(2, 2, error)
        IF error < 0 GOTO 120

; ================================================================
;
; This is a motor-following example based on motor position.  The two
; joints will track each other until SIGNAL 1001 is LOW.

    50  CALL sl.where(1, ref.pos, , error)
        IF error < 0 GOTO 120

        DO

            ; Move joint 2 of robot 1 (joint is assumed to be a
            ; translational joint).

            MOVE start:TRANS(0,(RANDOM*50))
            DO

                    ; Get the motor position of the joint 2.

                    CALL sl.where(2, cur.pos, , error)
                    IF error < 0 GOTO 120

                    ; Drive the second robot.

                    mtr.data[2] = ref.pos-cur.pos
                    ref.pos = cur.pos
                    mtr.data[1] = 6
                    CALL sl.mslew(1, mtr.data[], 1, FALSE, enc.pos[],
error)
            UNTIL STATE(2) == 2
        UNTIL NOT SIG(1001)

        ; Return second robot to original position.

        CALL sl.c.drive(2, 2, 1000, 50, start.pos, TRUE, error)
        IF error < 0 GOTO 120
        CALL sl.c.break(2, 2, error)
```

```
            IF error < 0 GOTO 120

; Disable ASYNCH mode.

   120      CALL sl.c.enable(2, 2, FALSE, FALSE, error)
            IF error < 0 THEN
                CALL sl.adv.diag(error)
            END

; Delete the Advanced Servo Library data structures.

            CALL sl.asl.cleanup()

            RETURN
.END
```

ex.j4.gainset( ) - Dynamic Tuning of Adept Joint 4 Example

This example program shows how to dynamically change the tuning on Joint 4 for an Adept robot.

```
.PROGRAM ex.j4.gainset()

; ABSTRACT:  Example routine that shows how to change the gain of
;       Joint 4 on an Adept robot on the fly.  In this example, the
;       Integral Gain is changed.
;
;       This example demonstrates how to dynamically change the tuning
;       of joint 4 on an Adept SCARA robot.  The hypothetical
;       application could be that the robot has to manipulate two
;       different parts with different end effectors, which have
;       significantly different weights.
;
;       You would need to know the tuning parameters in advance before
;       attempting to change the tuning dynamically.  These advance
;       values could be determined in one of two ways:
;
;           1.  By using the SPEC utility to plot and change the values
;               until you are satisified with the performance.
;
;           2.  By using the Advanced Servo Library to collect, analyze
;               and alter the values until you are satisfied with the
;               performance.
;
;       In either case, it is assumed that the values are known before
;       hand.
;
;       The example code below moves the robot to three different
;       locations:  loc.1, loc.2 and loc.3.  The gain for Joint 4 will
;       be altered between locations loc.2 and loc.3.  It will then be
;       reset back to the original factory value on the way back to
;       loc.1.
;
; INPUT PARM:   None
;
; OUTPUT PARM:  None
;
; DATA STRUCT:  loc.1   Global variables defining locations to which
;               loc.2     the robot wil be moved.  THESE LOCATIONS ARE
;               loc.3     NOT DEFINED BY THIS ROUTINE.  THEY MUST BE
;                         TAUGHT BEFORE EXECUTING THIS PROGRAM.
;
; SIDE EFFECTS: The gain for joint 4 of the Adept robot will be changed.
;               This may cause the motor to become unstable.  USE CAUTION!
;
; MISC:  This routine is for example purposes only.  You should
;        experiment with the SPEC parameters to obtain optimal joint
```

```
;        performance under different loads.
;
;* Copyright (c) 1996-97 by Adept Technology, Inc.

            AUTO error, mtr, orig.igain
            LOCAL stt[]

; Initialize AUTO values

            mtr = 4                                         ;Joint 4

; Initialize the ASL

            CALL sl.init(, stt[])
            IF stt[1] <> TRUE GOTO 100;If error exit routine

            ATTACH ()                           ;Attach the robot

;Read in the orginal factory setting for the Integral Gain.

            CALL sl.rw.tuning(mtr, FALSE, FALSE, , , , orig.igain, , , , , , error)
            IF error < 0 GOTO 100

            TYPE "Original Integral gain for Joint", mtr, " =", orig.igain

            TYPE /C1, "Moving to LOC.1 ..."
            MOVE loc.1                          ;Move to loc.1

            BREAK                               ;Wait for motion to complete

            TYPE /C1, "Moving to LOC.2 ..."
            MOVE loc.2                          ;Move to loc.2

            BREAK                               ;Wait for motion to complete

; Increase Integral Gain by 25%.

            CALL sl.rw.tuning(mtr, TRUE, FALSE, , , , orig.igain*1.25, , , , ,
, error)
            IF error < 0 GOTO 100

            TYPE "New Integral gain for Joint", mtr, " =", orig.igain*1.25

            TYPE /C1, "Moving to LOC.3 ..."
            MOVE loc.3                          ;Move to loc.3

; Wait until the motion to loc.3 is at least 85% complete.

            WAIT STATE(10) >= 85

; Reset the Integral Gain back to the original value

            CALL sl.rw.tuning(mtr, TRUE, FALSE, , , , orig.igain, , , , , , error)
            IF error < 0 GOTO 100
```

```
            MOVE loc.1                          ;Move back to loc.1

            BREAK                               ;Wait for motion to complete

            CALL sl.asl.cleanup()   ;Delete ASL globals from RAM

   100      RETURN
.END
```

ex.calibrate( ) - Motor Calibration Example

This example program shows how to calibrate a motor using the Advanced Servo Library routines.

```
.PROGRAM ex.calibrate(robot, $pwd, mtr, error)

; ABSTRACT:  This routine shows how to perform motor calibration for a
;       single motor on the specified robot.  High Power must be
;       enabled prior to calling this routine.
;
; INPUT PARMS:  robot   (Optional)  Robot number containing the motor
;                       that is to be calibrated.  Default is robot #1
;                       if undefined.
;               $pwd    (Optional)  Robot module password, if required.
;                       Default is NULL string if undefined.
;               mtr     Motor number on 'robot' to calibrate.
;
; OUTPUT PARMS: error   Success/error status of the motor calibration.
;                       Returns -401 (Undefined value) if 'mtr' argument
;                          is undefined.
;                       Returns -407 (Invalid argument) if selected robot
;                          does not contain motor number specified by
;                          'mtr'.
;                       Returns -604 (Robot power off) if high power is
;                          not enabled.
;                       Returns 0 or 1 if routine completes successfully.
;
; SIDE EFFECTS: Motor motion and loss of current calibration.
;
; NOTE: If this routine is used to calibrate all the motors on a given
;       robot, the system parameter NOT.CALIBRATED must be updated to
;       reflect the correct calibration status of the robot(s) in the
;       system.  This parameter informs V+ of the calibration status
;       of of the robot(s) in the system.  If this parameter is set
;       incorrectly the user application program will not be able to
;       ATTACH() to the robot.  The calling program must set this
;       parameter.  This routine does NOT modify the NOT.CALIBRATED
;       parameter.  See the V+ Language User's Guide for details.
;
; Copyright (c) 1997 by Adept Technology, Inc.

            AUTO DOUBLE j.status
            AUTO accel.time, dist, er.undefined, er.inv.arg, er.pwr.off
            AUTO er.no.zidx, srch.type
            AUTO REAL cal.dir, cal.method, cal.pos, cal.type, cur.pos
            AUTO REAL exp.dist, fine.speed, hrd.tol, srch.dist, srch.speed
            AUTO REAL states[6]
            AUTO $a, $pwd[15]
```

```
; Define error codes referenced by this program

            er.undefined = -401     ; *Undefined value*
            er.inv.arg = -407       ; *Invalid argument
            er.pwr.off = -604       ; *Robot power off*
            er.no.zidx = -1004      ; *No zero index*

            error = FALSE                       ;Assume success

            DETACH (0)                          ;Detach robot

; Select required robot

            IF NOT DEFINED(robot) THEN;If no robot was specified,
                robot = 1                       ; access the default robot
            END
            SELECT ROBOT = robot    ;Select the robot

; Make sure motor argument is defined and valid

            IF NOT DEFINED(mtr) THEN
                error = er.undefined            ; *Undefined value*
                GOTO 100
            END

            IF OUTSIDE(1,mtr,ID(3,10+robot)) THEN
                error = er.inv.arg              ; *Invalid argument*
                GOTO 100
            END

; Check High Power Enable status

            IF NOT SWITCH(POWER) THEN
                error = er.pwr.off              ; *Robot power off*
                GOTO 100                        ;Exit routine
            END

; Define the password(s) if required.

            IF DEFINED($pwd) THEN
                $pwd[robot] = $pwd  ;Use password passed in
            ELSE
                $pwd[robot] = ""    ;Assume no password
            END

; Initialize the Advanced Servo Library.

            TYPE "Initializing the Advanced Servo Library..."
            CALL sl.init($pwd[], status[])
            IF NOT status[robot] THEN
                TYPE "Advanced Servo Library initialization failure ", /S
                TYPE "for robot #", /I0, robot, "."
                PROMPT "Press 'ENTER' to exit routine.", $a
```

```
                GOTO 110                        ;Exit
            END

; Enable ASYNCH mode for the requested motor.

            TYPE "Enabling ASYNC mode..."
            CALL sl.enable(mtr, TRUE, FALSE, error)
            IF error < 0 THEN
                TYPE "Error entering ASYNC mode for motor", mtr, " : ", /S
                GOTO 100
            END

; Read the current calibration configuration from the servos for the
; requested motor.  This routine returns the following data:
;
;  cal.type:    0 = Calibrate using home sensor
;               1 = Calibrate using hard stop
;               2 = Calibrate to current position
;
;  cal.method:  0 = Calibrate to zero index
;               1 = Calibrate to home switch or hardstop
;
;  cal.dir:     0 = Calibrate to zero index moving OFF the sensor
;               1 = Calibrate to zero index moving ON the sensor

            CALL sl.rw.calcfg(mtr, FALSE, cal.type, cal.method, cal.dir,
error)
            IF error < 0 THEN
                TYPE "Error reading calibration configuration from servos for
motor ", mtr, ":"
                GOTO 100
            END

; Read calibration control parameters from V+.  This is the data entered in
; SPEC.

            CALL ex.get.cal.parm(mtr, srch.dist, cal.pos, srch.speed,
fine.speed, hrd.tol, $txt, error)
            IF error < 0 THEN
                TYPE $txt           ;Type out error text
                GOTO 100            ;Exit routine
            END

; Begin motion to calibration feature

            IF cal.type == 2 THEN   ;Calibrate to current position?

; Read current motor position

                CALL sl.read(mtr, sl.motpos, cur.pos, error)
                IF error < 0 THEN
                        TYPE "Error reading current motor position for motor
", mtr
                        GOTO 100
```

```
                END

; Set position reference and mark the motor calibrated.  (The sl.here &
; sl.adjust routines mark the motor calibrated as far as the servo's are
; concerned.)

                CALL sl.here(mtr, cur.pos, error)
                IF error < 0 THEN
                        TYPE "Error setting position reference for motor ",
mtr
                        GOTO 100
                END
            END

; Enable the amplifier.   The amp must be enabled before any motion can occur.

            CALL sl.amp(mtr, TRUE, error)
            IF error < 0 THEN
                TYPE "Error enabling the amplifier for motor ", mtr, "."
                GOTO 100
            END

            IF cal.type == 2 GOTO 110;Motor is now calibrated.  Exit.

; Initialize general calibration parms

            accel.time = 16         ;Acceleration time in milliseconds

            IF cal.type == 0 THEN   ;Calibrating to hard stop
                srch.type = 1       ;Stop at hard stop 'transition'
            ELSE
                srch.type = 0       ;Stop at home switch transition
            END

; Begin search for calibration feature (home switch or hard stop)

            CALL sl.calibrate(mtr, srch.dist, srch.speed, accel.time, FALSE,
srch.type, hrd.tol, states[], error)
            IF error < 0 THEN
                TYPE "Error beginning calibration motion for motor ", mtr, "."
                GOTO 100
            END

; Loop here checking calibration status until motor reaches calibration
; feature (hard stop or home sensor)

            WHILE states[0] <> 0 DO ;While motor trajectory is NOT done...

                CALL sl.calibrate(mtr, , , , , , , states[], error)
                IF error < 0 THEN
                        TYPE "Error polling calibration status for motor ",
mtr, "."
                        GOTO 100
                END
```

```
            END

; If we made it this far, the motor should be stopped at the calibration
; feature (either the hard stop, or the home switch)
;
; What we do next depends on the calibration method & type.
; If we are calibrating to the hard stop:
;       1) Drive away from the hard stop looking for the Z-Index, if necessary
;       2) Update the encoder register with the motor position at the hard
stop.
;
; If we are calibrating to the home switch:
;       1) Find the edge of the home switch
;       2) Drive in the proper direction to the zero index, if necessary
;       3) Update the encoder register with the motor position at the index.

            IF cal.method THEN;Calibrate to feature (hard stop or home switch)
                                                ;No Z-Index search.

; Update position reference

                CALL sl.here(mtr, cal.pos, error)
                IF error < 0 THEN
                        TYPE "Error setting position reference for motor ",
mtr, "."
                        GOTO 100
                END

            ELSE                    ;Calibrate to Z-Index

; Read expected distance from cal feature to 1st Z-Index

                CALL sl.read(mtr, sl.homin4, exp.dist, error)
                IF error < 0 THEN
                        TYPE "Error reading expected distance to Z-Index for
motor ", mtr, "."
                        GOTO 100
                END

; Compute direction of fine search plus a 1000 count tolerance

                dist = -SIGN(srch.dist)*(exp.dist+1000)

                IF cal.type == 0 THEN           ;Find edge of home switch 1st.

; Read current state of home switch

                        CALL sl.status(mtr, j.status, error)
                        IF error < 0 THEN
                            TYPE "Error Reading Home Switch state for motor ",
mtr, "."
                            GOTO 100
                        END
```

```
; Get current state of home switch (results in: 0 or -1)

                        home.sw.state = (j.status BAND ^H1) == 1

; Drive motor towards edge of home switch

                        CALL sl.drive(mtr, 8, fine.speed, dist, FALSE, error)
                        IF error < 0 THEN
                            TYPE "Error Driving motor ", mtr, "."
                            GOTO 100
                        END

; Poll for change in home switch state

                        DO
                            CALL sl.status(mtr, j.status, error)
                            IF error < 0 THEN
                                    TYPE "Error Reading Home Switch state for
motor ", mtr, "."
                                    GOTO 100
                            END
                        UNTIL ((j.status BAND ^H1) == 1) <> home.sw.state

; ABORT motion

                        CALL sl.stop(mtr, error)
                        IF error < 0 THEN
                            TYPE "Error Aborting motion for motor ", mtr, "."
                            GOTO 100
                        END
                END

; Begin motion towards Z-Index

                CALL sl.drive(mtr, 8, fine.speed, dist, FALSE, error)
                IF error < 0 THEN
                        TYPE "Error Driving motor ", mtr, "."
                        GOTO 100
                END

; Poll status register until 1st Z-index is seen or motion completes.

                DO
                        CALL sl.status(mtr, j.status, error)
                        IF error < 0 THEN
                            TYPE "Error Reading Joint status for motor ", mtr,
"."
                            GOTO 100
                        END
                UNTIL ((j.status BAND ^H2000000) == ^H2000000) OR (j.status
BAND ^H10000 == ^H10000)

; ABORT motion
```

```
                CALL sl.stop(mtr, error)
                IF error < 0 THEN
                        TYPE "Error Aborting motion for motor ", mtr, "."
                        GOTO 100
                END

; Wait for motor to stop

                CALL sl.break(mtr, error)
                IF error < 0 THEN
                        TYPE "Error Aborting motion for motor ", mtr, "."
                        GOTO 100
                END

; Check to see if we actually saw the Z-Index.  Otherwise, return an error.

                IF (j.status BAND ^H2000000) <> ^H2000000 THEN
                        TYPE "ERROR:  No Zero index found for motor ", mtr
                        error = er.no.zidx;No Zero Index found
                        GOTO 100
                END

; Update encoder register

                CALL sl.here(mtr, cal.pos, error)
                IF error < 0 THEN
                        TYPE "Error updating encoder register for motor ",
mtr, "."
                        GOTO 100
                END

            END

; Re-enable SYNC mode for this motor

            CALL sl.enable(mtr, FALSE, FALSE, error)
            IF error < 0 THEN
                TYPE "Error re-enabling SYNC mode for motor ", mtr
                GOTO 100
            END

            GOTO 110    ;Exit skipping error reporting

; Display V+ or Servo error information

   100      CALL sl.adv.diag(error)

; Exit

   110      RETURN
.END
```

```
.PROGRAM ex.get.cal.parm(mtr, srch.dist, cal.pos, srch.speed, fine.speed,
hrd.tol, $txt, error)

; ABSTRACT: This routine assists the ex.calibrate routine.  It reads
;       all the necessary calibration parameters.
;
; INPUT PARMS:  mtr     Motor number
;
; OUTPUT PARMS: srch.dist    Search distance in encoder counts
;               cal.pos      Motor position at calibration feature
;                               (home switch or hard stop)
;               srch.speed   Search speed
;               fine.speed   Fine search speed
;               hrd.tol      Home sensor width or hard stop tolerance
;               $txt         If an error occurs, text string that
;                               contains information about what
;                               parameter caused the error.
;               error        Success/failure status of this routine.
;
; SIDE EFFECTS: None
;
; Copyright (c) 1997 by Adept Technology, Inc.

; Initialize error outputs

            $txt = ""               ;No text to display
            error = FALSE           ;Assume success

; Read Maximum search distance

            CALL sl.read(mtr, sl.homin1, srch.dist, error)
            IF error < 0 THEN
                $txt = "Error reading Search Distance from servos for motor "
                GOTO 100
            END

; Read Calibration Position at 1st index or other feature

            CALL sl.read(mtr, sl.homin2, cal.pos, error)
            IF error < 0 THEN
                $txt = "Error reading Calibration Position from servos for
motor "
                GOTO 100
            END

; Read Search speed

            CALL sl.read(mtr, sl.homin6, srch.speed, error)
            IF error < 0 THEN
                $txt = "Error reading Search Speed from servos for motor "
                GOTO 100
            END

; Read Fine Search speed
```

```
            CALL sl.read(mtr, sl.homin7, fine.speed, error)
            IF error < 0 THEN
                $txt = "Error reading Fine Search Speed from servos for motor
"
                GOTO 100
            END

; Read Home sensor width or hard stop tolerance.  The meaning depends on the
; calibration method

            CALL sl.read(mtr, sl.homin3, hrd.tol, error)
            IF error < 0 THEN
                $txt = "Error reading Hard Stop Tolerance from servos for
motor "
            END

   100      IF LEN($txt) THEN;If an error occured...
                $txt = $txt+$ENCODE(mtr," .")  ;Encode motor number to error
txt
            END

            RETURN
.END
```

ex.free( ) - FREE Mode Example

This example program shows how to make Joint 4 (RZ) a compliant joint to ease part insertion.

```
.PROGRAM ex.free(robot, $password, error)

; ABSTRACT:  This is an example routine that will demonstrate Joint 4
;       compliance in order to successfully insert a chamfered, keyed
;       part into a hole.
;
;           This routine assumes that two locations are ALREADY defined,
;                       PICK
;                       PLACE
;
;           The robot will perform the following:
;
;                       1) Approach the PICK location
;                       2) Aquire the part
;                       3) Depart PICK location
;                       4) Approach the PLACE location
;                       5) Place J4 into FREE mode
;                       6) Move to PLACE location
;                       7) Release the part
;                       8) Depart PLACE location
;                       9) Take J4 out of FREE mode
;
; INPUT PARMS:  robot      (Optional) Number of robot to move.
;                             Default is robot #1.
;               $password  (Optional) Robot module password to
;                             access SPEC data.  May be undefined.
;
; OUTPUT PARMS: error   Standard success/error value
;
; SIDE EFFECTS: None
;
; Copyright (c) 1997 by Adept Technology, Inc.

; Select specified robot.


            IF NOT DEFINED(robot) THEN          ;If no robot was specified,
                robot = 1                       ; access the default robot
            END

            IF SELECT(ROBOT) <> robot THEN      ;Different robot selected?
                DETACH () ;Robot must be detached before selecting new one
                SELECT ROBOT = robot            ;Select the robot
            END
```

```
; Define the password(s) if required.

            IF DEFINED($password) THEN
                $password[robot] = $password    ;Use password passed in
            ELSE
                $password[robot] = ""           ;Assume no password
            END

; Initialize things.

            TYPE "Initializing the Advanced Servo Library..."

            CALL sl.init($password[], status[])
            IF status[robot] <> -1 THEN
                TYPE "Advanced Servo Library initialization failure ", /S
                TYPE "for robot #", /I0, robot, "."
                GOTO 100                        ;Exit
            END

            OPEN                    ;Open gripper

; Approach PICK location

            APPRO pick, 50
            BREAK

; Move to PICK location


            MOVE pick
            BREAK

            CLOSE                   ;Aquire the part
            WAIT.EVENT , 0.3        ;Wait for mechanical motion to complete

; Depart PICK location

            DEPART 50

; Approach PLACE location

            APPRO place, 50

; Place J4 into FREE mode.  The first argument for the sl.free routine is a
; bitmask indicating which motors to place into free mode.  There is one bit
; per motor.  Bit 1 (LSB) is for motor one, bit 2 is for motor 2 etc...

            CALL sl.free(^B1000, TRUE, error)
            IF error < 0 GOTO 100

            SPEED 25                ;Reduce speed to 25%
```

```
; Move to PLACE location

            MOVE place
            BREAK

            OPEN                    ;Release part
            WAIT.EVENT , 0.3        ;Wait for mechanical motion to complete

; Depart PLACE location

            DEPART 50

; Place J4 back into position servo mode

            CALL sl.free(^B1000, FALSE, error)
            IF error < 0 GOTO 100


   100      IF error < 0 THEN
                CALL sl.adv.diag(error)
            END
            RETURN
.END
```

Index

A

access, Adept robot 11
Adept joint 4 example, dynamic tuning of 182
Adept robot access 11
Advanced Servo Library example programs 163
asynchronous mode, synchronous vs. 19
axis following examples 178

B

basis, reading/writing on a channel 17
basis, reading/writing on a motor-number 17
buffer example, setpoint 172
buffer, routines for using the setpoint 15

C

channel basis, reading/writing on a 17
channel numbers vs. motor numbers 17
collecting servo data, routines for 13
compatibility 11

D

DAC output routines, encoder and 14
data monitoring example, joint motion 168
data, routines for collecting servo 13
descriptions of library routines 21
descriptions, parameter 17
dynamic tuning of Adept joint 4 example 182

E

encoder and DAC output routines 14
errors 20
example
  dynamic tuning of Adept joint 4 182
  joint motion data monitoring 168
  programs, Advanced Servo Library 163
  robot motion 164
  routines 16
  setpoint buffer 172
examples
  axis following 178

I

initialization 18
initialization and other utility routines 12
installation 12
Introduction 10

J

joint 4 example, dynamic tuning of Adept 182
joint motion data monitoring example 168

L

library
  example programs, Advanced Servo 163
  routines, descriptions of 21
  routines, summary of 12

M

miscellaneous routines 14
mode, synchronous vs. asynchronous 19
module specific routines, split axis gantry 14
monitoring example, joint motion data 168
motion
  data monitoring example, joint 168
  example, robot 164
  routines, robot 13
motor numbers, channel numbers vs. 17
motor-number basis, reading/writing on a 17

N

numbers, channel numbers vs. motor 17

O

other utility routines, initialization and 12
output routines, encoder and DAC 14
overview 9

P

parameter descriptions 17
parameters, routines for reading/writing
servo 15
password protection 18
programs, Advanced Servo Library
example 163
protection, password 18
publications, related 12

R

reading/writing
on a channel basis 17
on a motor-number basis 17
servo parameters, routines for 15
related publications 12
robot
access, Adept 11
motion
example 164
routines 13
routine
sl.adjust( ) 23
sl.adv.diag( ) 24
sl.amp( ) 25
sl.asl.cleanup( ) 26
sl.brake( ) 27
sl.break( ) 28
sl.c.adjust( ) 29
sl.c.amp( ) 30
sl.c.brake( ) 31
sl.c.break( ) 32
sl.c.drive( ) 33
sl.c.enable( ) 34
sl.c.getdac( ) 37
sl.c.here( ) 38
sl.c.pcocfg( ) 39
sl.c.read( ) 42
sl.c.setdac( ) 43
sl.c.status( ) 44
sl.c.stop( ) 46
sl.c.write( ) 50
sl.calibrate( ) 52
sl.defdata( ) 59
sl.deftrig( ) 61
sl.defwave( ) 63
sl.drive( ) 64
sl.enable( ) 66
sl.exttrig( ) 67
sl.getdac( ) 70
sl.getdata( ) 71
sl.hand.ctrl( ) 73
sl.here( ) 74
sl.init( ) 75
sl.jnt.to.mtr( ) 77
sl.listdata( ) 78
sl.mslew( ) 79
sl.mtr.to.jnt( ) 82
sl.over.travel( ) 83
sl.pcocfg( ) 84
sl.read( ) 87
sl.read.file( ) 94
sl.rw.accel( ) 99
sl.rw.amask( ) 100
sl.rw.bcklsh( ) 101
sl.rw.cal.file( ) 102
sl.rw.calcfg( ) 103
sl.rw.collision( ) 104
sl.rw.comp( ) 110
sl.rw.cspd( ) 112
sl.rw.ctol( ) 113
sl.rw.ftol( ) 116
sl.rw.hand( ) 117
sl.rw.id.data( ) 120
sl.rw.jaccel( ) 121
sl.rw.jmap( ) 122
sl.rw.link( ) 124
sl.rw.maxjspd( ) 128
sl.rw.mspd( ) 129
sl.rw.mtic( ) 130
sl.rw.rollover( ) 133
sl.rw.s_crv( ) 139
sl.rw.scale( ) 134
sl.rw.softstops( ) 137
sl.rw.tooloff( ) 140
sl.set.gan.splt( ) 143
sl.setdac( ) 144
sl.setpt.cfg( ) 145
sl.setpt.fill( ) 147

sl.slave.mode( ) 148
sl.slew( ) 149
sl.spare( ) 151
sl.status( ) 152
sl.stop( ) 154
sl.version( ) 155
sl.wave.trig( ) 156
sl.write( ) 158
sl.wrt.dat.file( ) 159
sl.wrt.sys.file( ) 161
routines
descriptions of library 21
encoder and DAC output 14
example 16
for collecting servo data 13
for reading/writing servo parameters 15
for using the setpoint buffer 15
initialization and other utility 12
misc. 14
robot motion 13
split axis gantry module specific 14
status 13
summary of library 12

S

servo data, routines for collecting 13
Servo Library example programs, Advanced 163
servo parameters, routines for reading/writing 15
setpoint buffer
example 172
routines for using the 15
sl.adjust( ) routine 23
sl.adv.diag( ) routine 24
sl.amp( ) routine 25
sl.asl.cleanup( ) routine 26
sl.brake( ) routine 27
sl.break( ) routine 28
sl.c.adjust( ) routine 29
sl.c.amp( ) routine 30
sl.c.brake( ) routine 31
sl.c.break( ) routine 32
sl.c.drive( ) routine 33
sl.c.enable( ) routine 34
sl.c.getdac( ) routine 37
sl.c.here( ) routine 38
sl.c.pcocfg( ) routine 39
sl.c.read( ) routine 42
sl.c.setdac( ) routine 43
sl.c.status( ) routine 44
sl.c.stop( ) routine 46
sl.c.write( ) routine 50
sl.calibrate( ) routine 52
sl.defdata( ) routine 59
sl.deftrig( ) routine 61
sl.defwave( ) routine 63
sl.drive( ) routine 64
sl.enable( ) routine 66
sl.exttrig( ) routine 67
sl.getdac( ) routine 70
sl.getdata( ) routine 71
sl.hand.ctrl( ) routine 73
sl.here( ) routine 74
sl.init( ) routine 75
sl.jnt.to.mtr( ) routine 77
sl.listdata( ) routine 78
sl.mslew( ) routine 79
sl.mtr.to.jnt( ) routine 82
sl.over.travel( ) routine 83
sl.pcocfg( ) routine 84
sl.read( ) routine 87
sl.read.file( ) routine 94
sl.rw.accel( ) routine 99
sl.rw.amask( ) routine 100
sl.rw.bcklsh( ) routine 101
sl.rw.cal.file( ) routine 102
sl.rw.calcfg( ) routine 103
sl.rw.collision( ) routine 104
sl.rw.comp( ) routine 110
sl.rw.cspd( ) routine 112
sl.rw.ctol( ) routine 113
sl.rw.ftol( ) routine 116
sl.rw.hand( ) routine 117
sl.rw.id.data( ) routine 120
sl.rw.jaccel( ) routine 121
sl.rw.jmap( ) routine 122
sl.rw.link( ) routine 124
sl.rw.maxjspd( ) routine 128
sl.rw.mspd( ) routine 129
sl.rw.mtic( ) routine 130
sl.rw.rollover( ) routine 133
sl.rw.s_crv( ) routine 139
sl.rw.scale( ) routine 134
sl.rw.softstops( ) routine 137
sl.rw.tooloff( ) routine 140
sl.set.gan.splt( ) routine 143

sl.setdac( ) routine 144
sl.setpt.cfg( ) routine 145
sl.setpt.fill( ) routine 147
sl.slave.mode( ) routine 148
sl.slew( ) routine 149
sl.spare( ) routine 151
sl.status( ) routine 152
sl.stop( ) routine 154
sl.version( ) routine 155
sl.wave.trig( ) routine 156
sl.write( ) routine 158
sl.wrt.dat.file( ) routine 159
sl.wrt.sys.file( ) routine 161
split axis gantry module specific routines 14
status routines 13
summary of library routines 12
synchronous vs. asynchronous mode 19

T

tuning of Adept joint 4 example, dynamic 182

U

using the setpoint buffer, routines for 15
utility routines, initialization and other 12

Adept User's Manual Comment Form

We have provided this form to allow you to make comments about this manual, to point out any mistakes you may find, or to offer suggestions about information you want to see added to the manual. We review and revise user's manuals on a regular basis, and any comments or feedback you send us will be given serious consideration. Thank you for your input.

NAME_____ DATE_____

COMPANY _____

ADDRESS_____

PHONE_____

MANUAL TITLE: _____

PART NUMBER and REV level:_____

COMMENTS:

_____

_____

_____

_____

_____

_____

_____

_____

_____

MAIL TO: Adept Technology, Inc.
Technical Publications Dept.
11133 Kenwood Rd.
Cincinnati, OH 45242

FAX: (513) 792-0274